프로그램 합성 기반 프로그램 최적화

이 우 석

한양대학교 에리카 컴퓨터학부

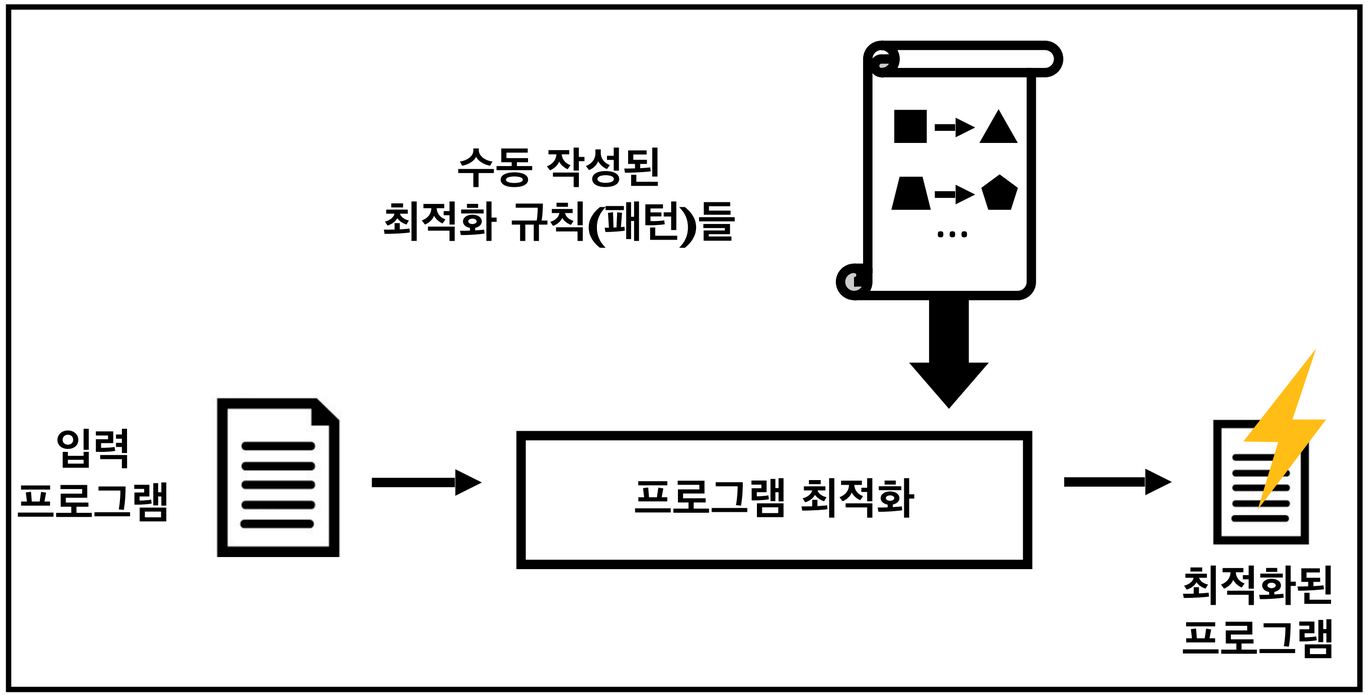
2024. 5. 24 @ POSTECH

발표자 소개

- 소속 : 한양대학교 에리카 컴퓨터학부 조교수 (2018.9 ~)
- 전공 : 프로그래밍언어, 프로그램 분석, 프로그램 자동 생성
- 이력 :
  - 2012 UC Berkeley 방문연구원
  - 2016 서울대학교 컴퓨터공학 박사 (지도교수: 이광근)
  - 2016-2017 Georgia Tech 박사후연구원
  - 2017-2018 University of Pennsylvania 박사후연구원

동기

- 프로그램 최적화
  - 더 나은 프로그램 (계산비용 등)으로 변환
  - 변환규칙(예: $x + 0 \rightarrow x$) 반복 적용



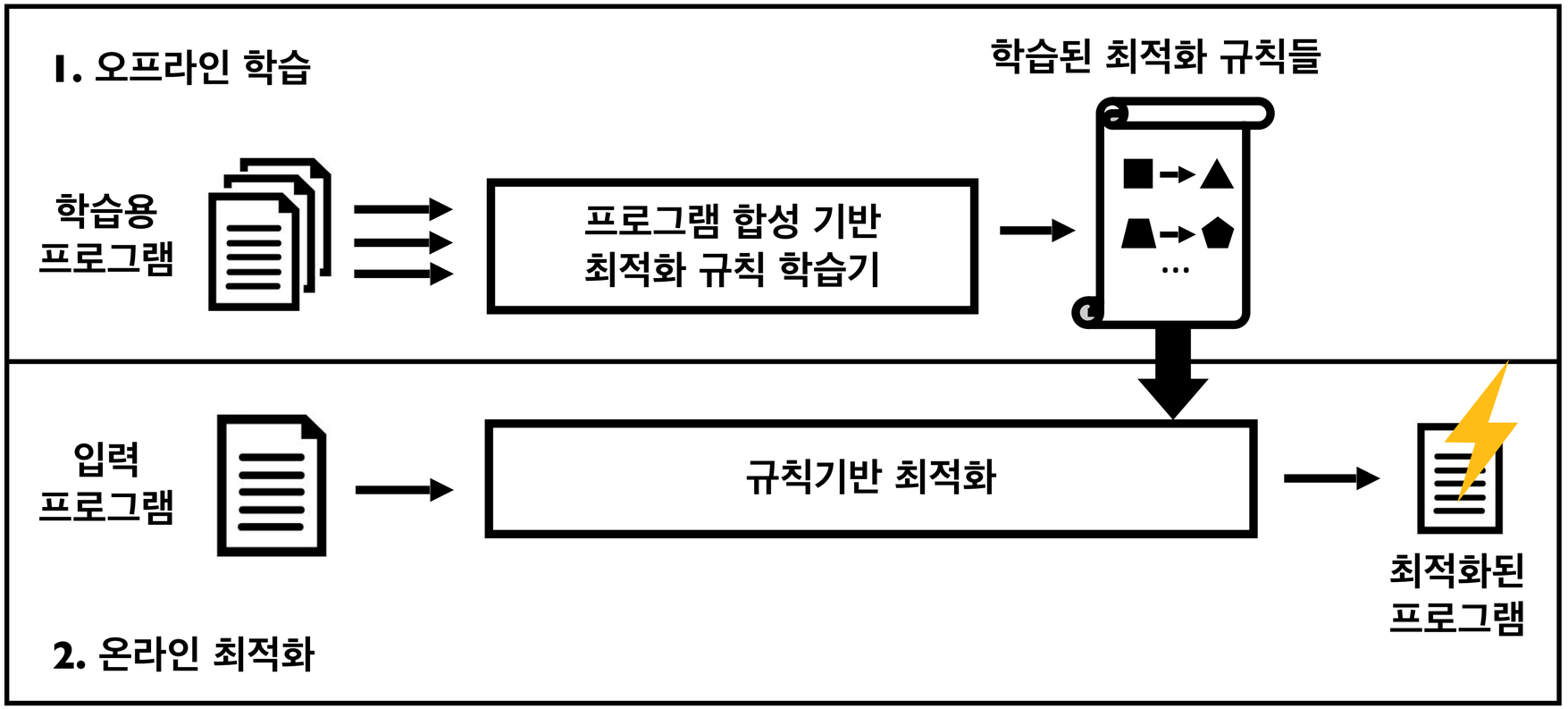
- 기존의 변환규칙
  - 전문가가 수동 작성
  - 제한된 규칙으로 탐색범위 제한됨
- 기존의 규칙 적용순서
  - 전문가가 정한 순서(휴리스틱) 따름
  - 최적해 놓침

해결책

최적화 규칙 발견(프로그램 합성*Program synthesis*) + 찾은 규칙 잘 적용하기(식 다시쓰기*Term rewriting*)
+ 모든 가능한 규칙 적용 순서 고려하기 (동일식 모으기*Equality saturation*)

규칙 발견이 오래 걸리는 경우

: 오프라인 학습
+ 온라인 최적화

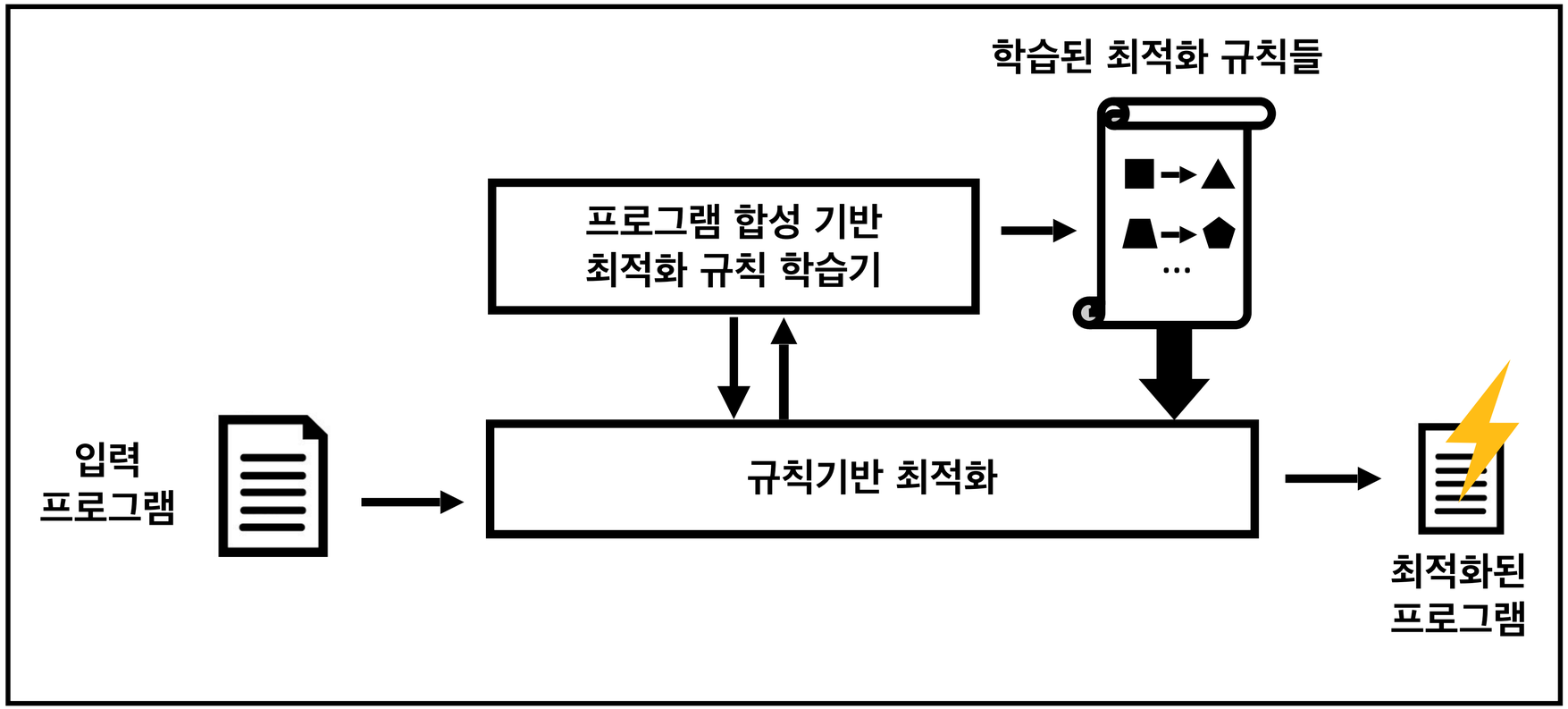


해결책

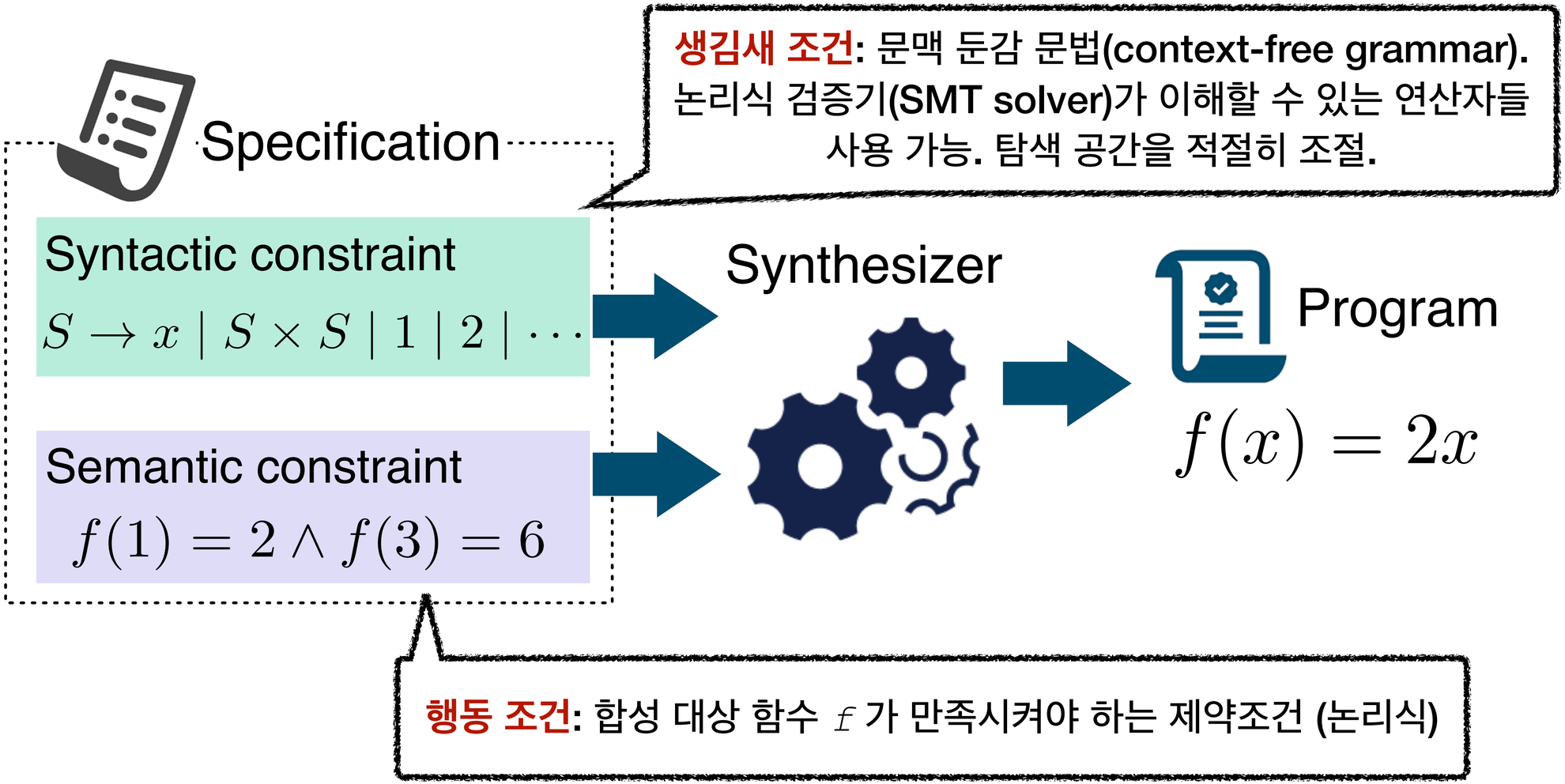
최적화 규칙 발견(프로그램 합성*Program synthesis*) + 찾은 규칙 잘 적용하기(식 다시쓰기*Term rewriting*) + 모든 가능한 규칙 적용 순서 고려하기 (동일식 모으기*Equality saturation*)

규칙 발견이 빨리되는 경우

: 온라인 학습 + 최적화

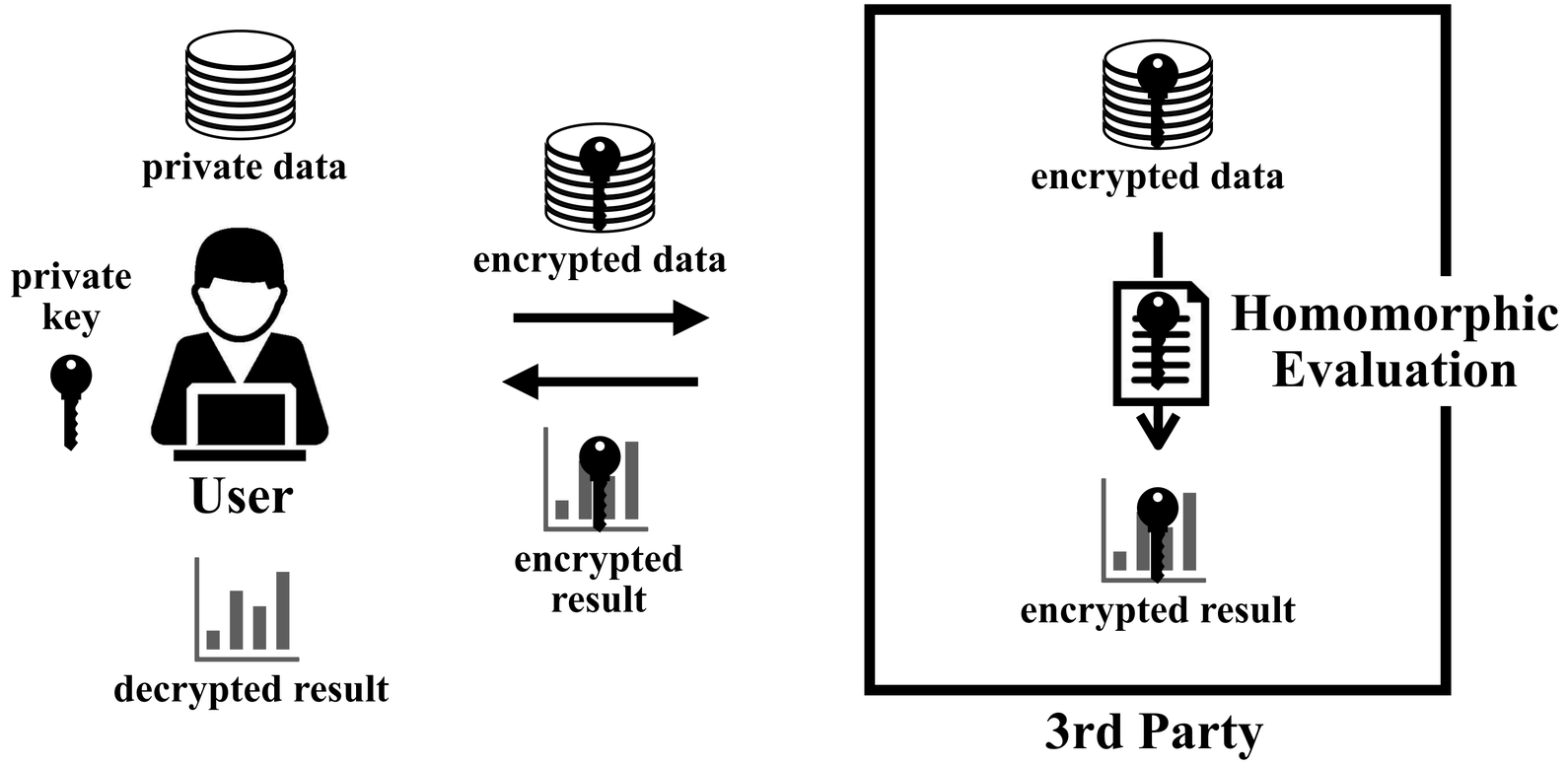


핵심 기반 기술 : 프로그램 합성



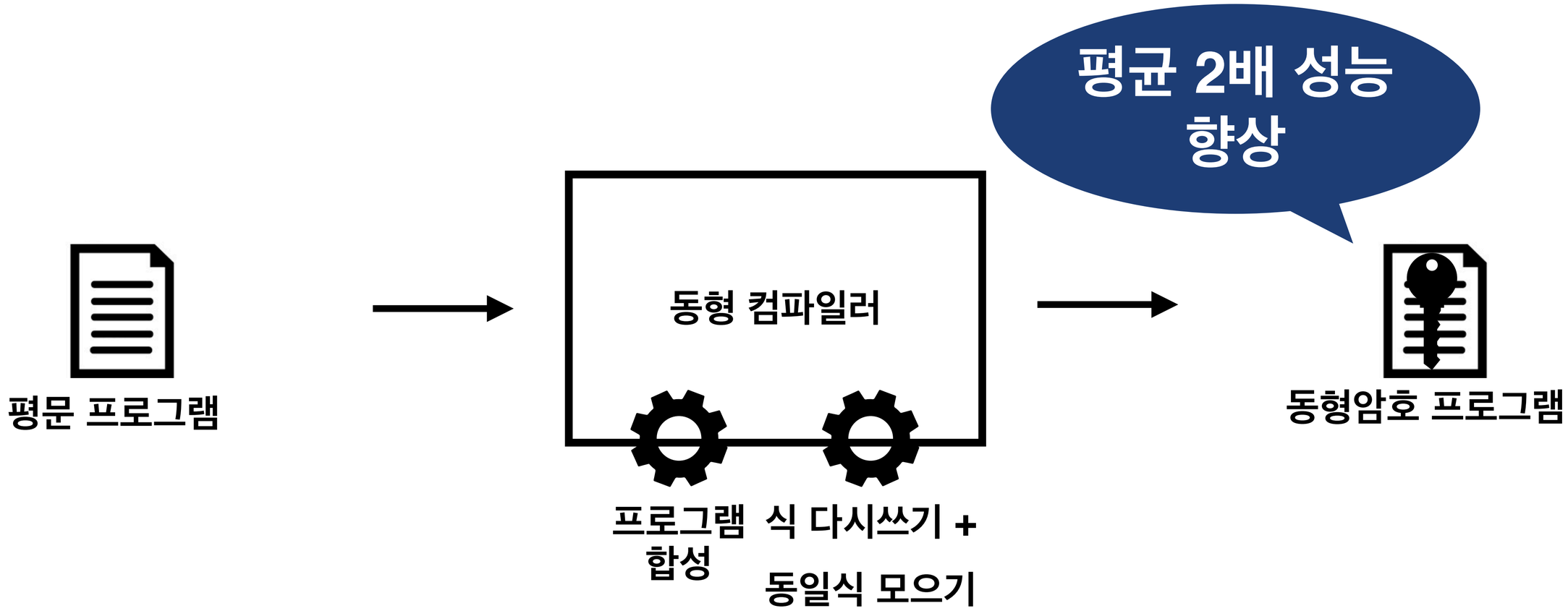
사례 1: 동형암호 컴파일러 (1/2)

- 동형암호: 암호화된 데이터 위에서 모든 연산 수행가능
- 개인정보 누출 원천차단
- 민감한 개인정보 보관/가공 아웃소싱 가능



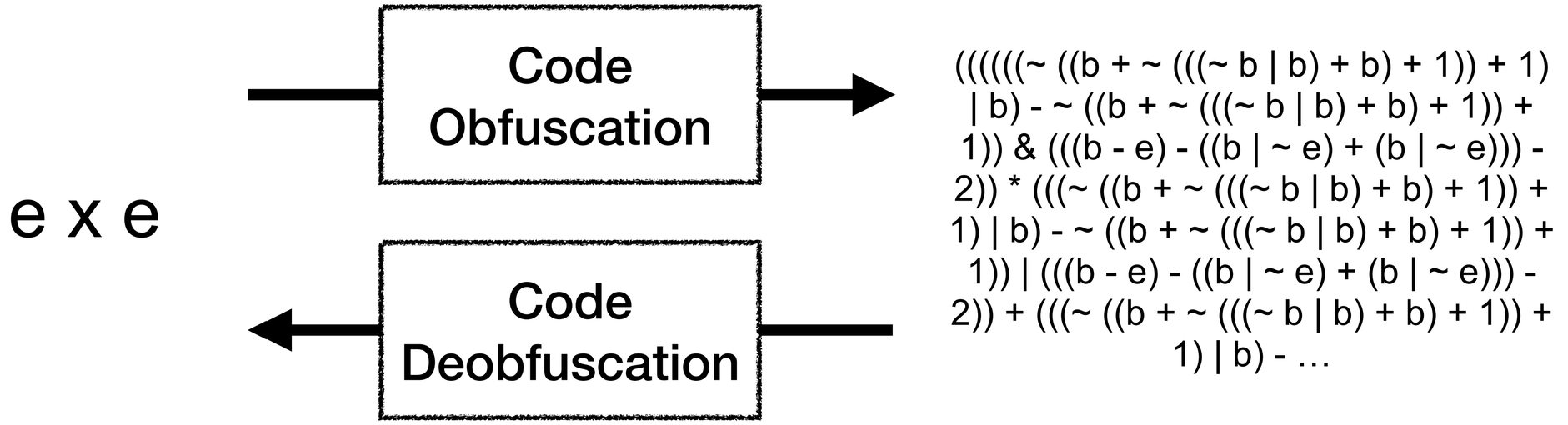
사례 1: 동형암호 컴파일러 (2/2)

- 동형암호 프로그램 자동생성
- 수동 작성된 최적화 규칙을 쓰는 컴파일러 보다 최적화 효과 우월



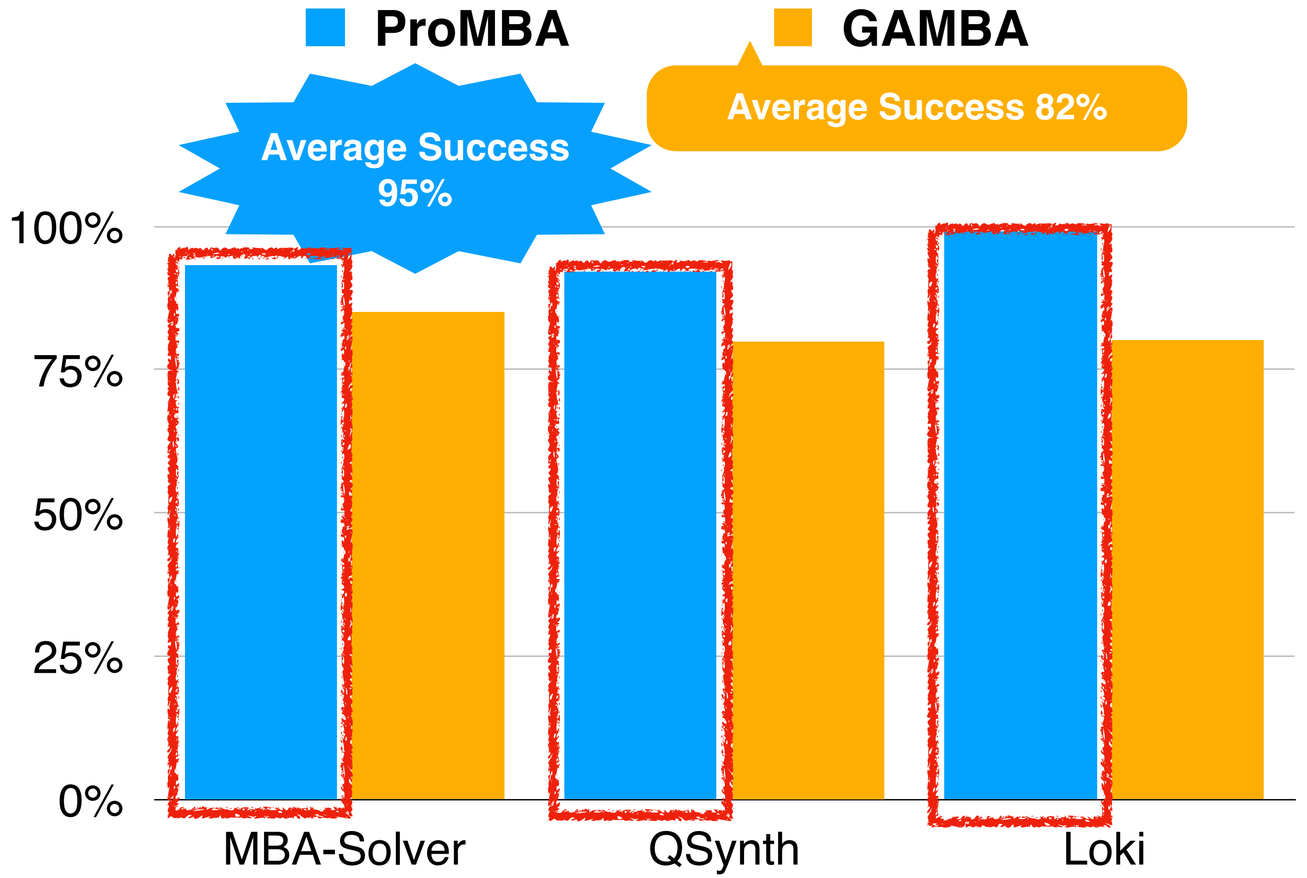
사례 2: 비트연산식 난독화 해제 (1/2)

- 난독화*Obfuscation*: 의미는 같으나 복잡하게 변환
  - 목적: 악성웨어 탐지 회피 😈 지적재산권 보호 😇
- 역난독화*Deobfuscation*: 난독화된 코드 단순화



사례 2: 비트연산식 난독화 해제 (2/2)

- 역난독화 성공 : 원본 식, 혹은 더 작은 식으로 단순화
- 전문가 알고리즘과 휴리스틱을 쓴 경우보다 높은 역난독화 성공률



논문들

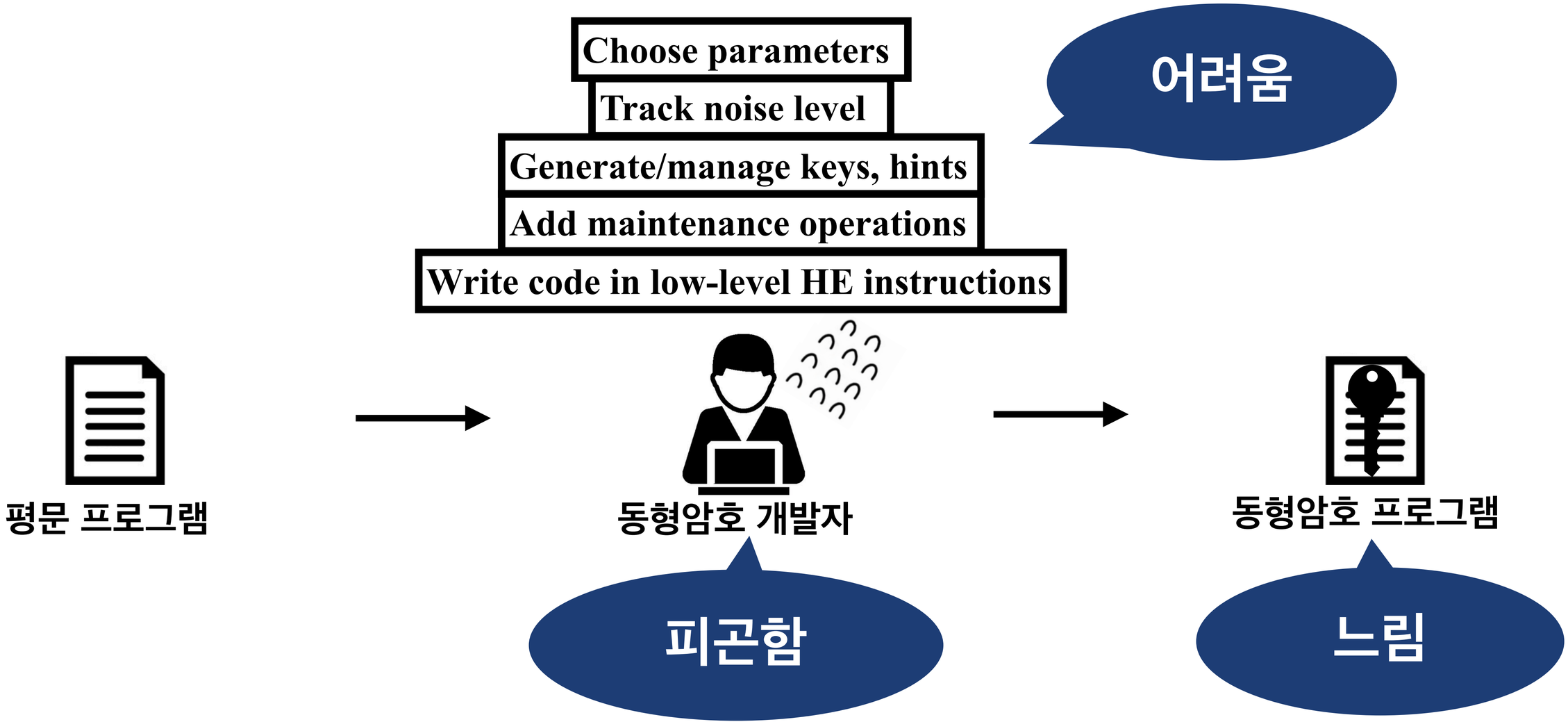
- 최적화 사례 1 : 프로그램 합성 기반 동형암호 최적화
  - Dongkwon Lee, Woosuk Lee, Hakjoo Oh and Kwangkeun Yi, *Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Time-Bounded Exhaustive Search,* **ACM TOPLAS 2023**
  - Dongkwon Lee, Woosuk Lee, Hakjoo Oh and Kwangkeun Yi, *Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting,* **ACM PLDI 2020**
- 최적화 사례 2: 프로그램 합성 기반 프로그램 역난독화
  - Jaehyung Lee and Woosuk Lee, *Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting*, **ACM CCS 2023**
  - Jaehyung Lee, Seoksu Lee, Eunsun Cho and Woosuk Lee, *Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Equality Saturation*, **IEEE TDSC (Submitted)**
- 기반 원천 기술 : 고성능 프로그램 합성 기술
  - Yongho Yoon, Woosuk Lee, and Kwangkeun Yi, Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. **ACM PLDI 2023**

차례

- **최적화 사례 1 : 프로그램 합성 기반 동형암호 최적화**
  - Dongkwon Lee, Woosuk Lee, Hakjoo Oh and Kwangkeun Yi, *Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Time-Bounded Exhaustive Search,* **ACM TOPLAS 2023**
  - Dongkwon Lee, Woosuk Lee, Hakjoo Oh and Kwangkeun Yi, *Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting,* **ACM PLDI 2020**
- 최적화 사례 2 : 프로그램 합성 기반 프로그램 역난독화
- 두 사례로 부터 관찰한 것들
- 기반 원천 기술 : 고성능 프로그램 합성 기술

동형암호 기술(Homomorphic Encrytion) (1/2)

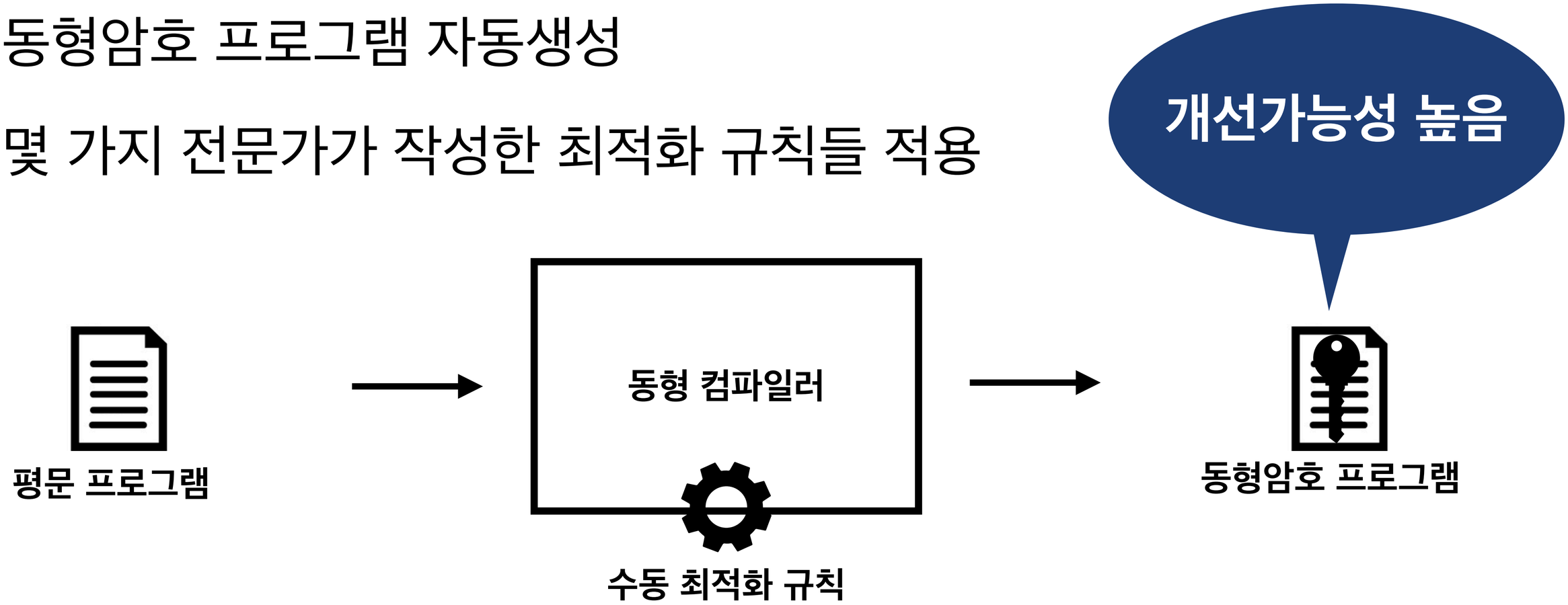
동형암호 프로그램 작성하기



동형암호 기술(Homomorphic Encrytion) (2/2)

동형암호 컴파일러 기존기술

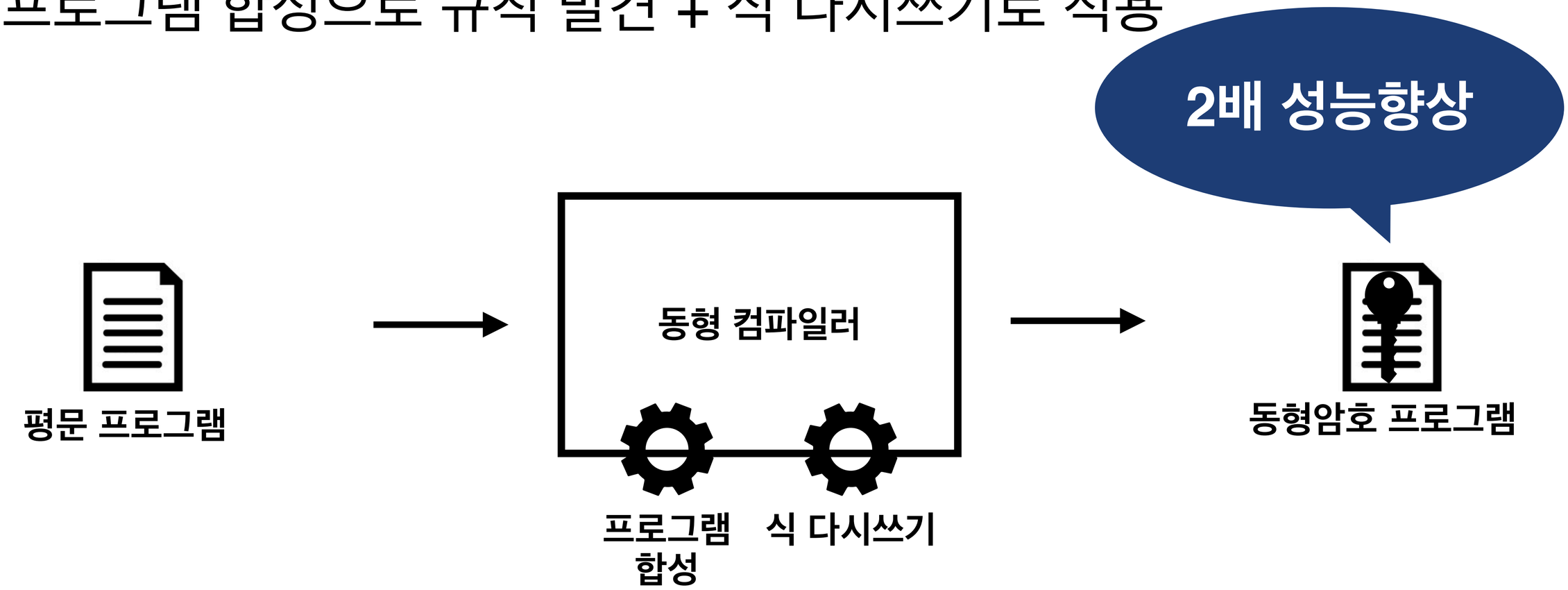
- 동형암호 프로그램 자동생성
- 몇 가지 전문가가 작성한 최적화 규칙들 적용



동형암호 컴파일러 개선점

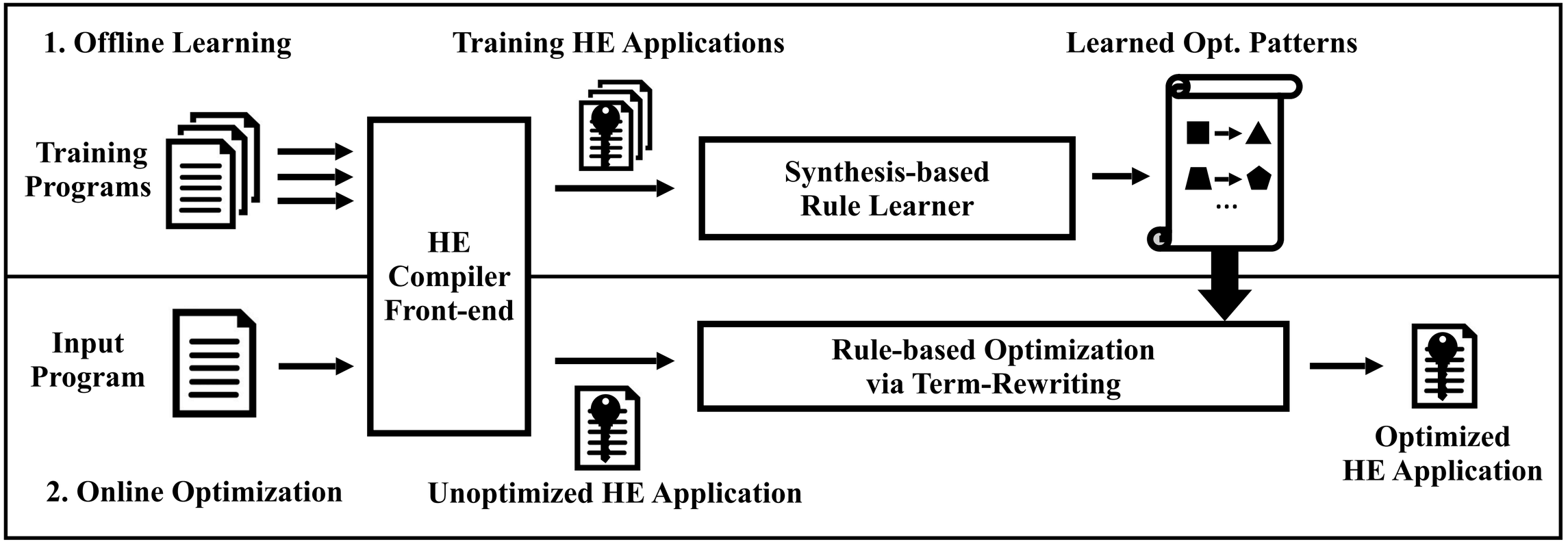
자동탐색 기반 동형암호 최적화 틀 제안

- 동형암호 프로그램 자동생성
- 프로그램 합성으로 규칙 발견 + 식 다시쓰기로 적용



Lobster

- 최적화 규칙 미리 찾기(프로그램 합성) + 찾은 규칙 잘 적용하기(식 다시쓰기)



간단한 동형암호

- Based on approximate common divisor problem
- p : 비밀키 (정수)
- q : 랜덤 수
- r ($\ll |p|$) : 랜덤 수 (**교란값**. 보안 강화 목적)
- 0과 1을 암호화 / 복호화하는 함수

$$Enc_p(\ \mu \in \{0,1\}\ ) = pq + 2r + \mu$$
$$Dec_p(c) = (c \bmod p) \bmod 2$$
$$Dec_p(Enc_p(\mu)) = Dec_p(pq + 2r + \mu) = \mu$$

- 암호문 $\underline{\mu_i} \leftarrow Enc_p(\mu_i)$에 대해 다음 성립

$$Dec_p(\underline{\mu_1} + \underline{\mu_2}) = \mu_1 + \mu_2$$
$$Dec_p(\underline{\mu_1} \times \underline{\mu_2}) = \mu_1 \times \mu_2$$

- 0과 1만 있는 세상에서 덧셈(XOR)과 곱셈(AND)를 암호화된 채로 수행 가능
  즉, 모든 논리 회로 암호화된 채로 실행 가능

성능의 걸림돌 — 증가하는 교란값*Noise*

- 교란값이 연산 중 증가
- 암호문 $\underline{\mu_i} = pq_i + 2r_i + \mu_i$ 에 대해서

$$\underline{\mu_1} + \underline{\mu_2} = p(q_1 + q_2) + \boxed{2(r_1 + r_2) + (\mu_1 + \mu_2)} \quad \text{2배 증가}$$

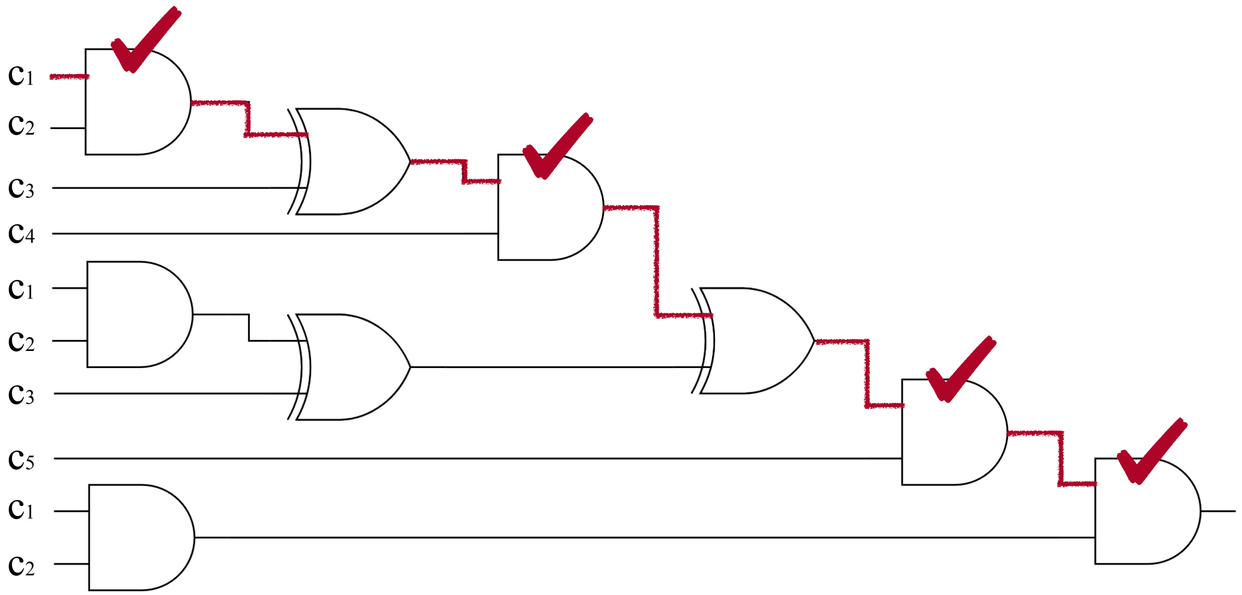
$$\underline{\mu_1} \times \underline{\mu_2} = p(pq_1q_2 + \cdots) + \boxed{2(2r_1r_2 + r_1\mu_2 + r_2\mu_1) + (\mu_1 \times \mu_2)} \quad \text{제곱 증가}$$

교란값

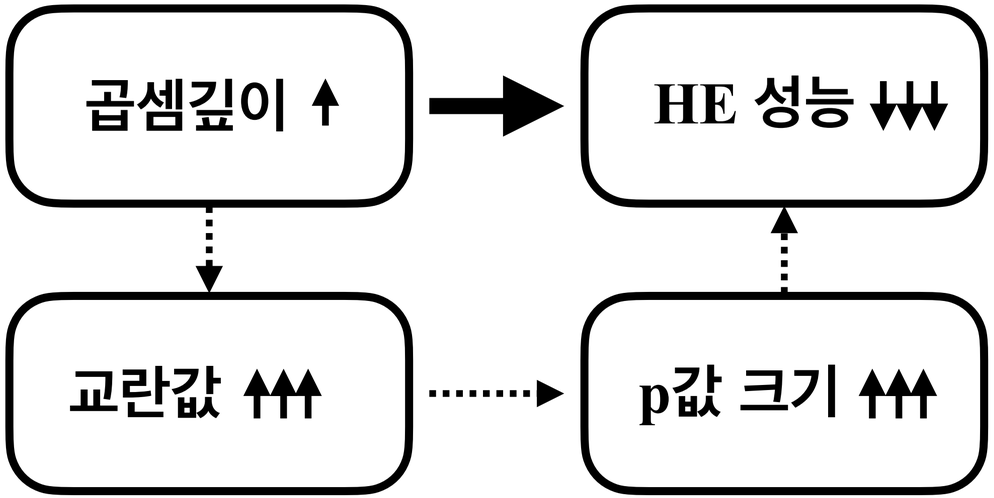
- **만약 (교란값 $> p$) 면 복호화 시 결과가 틀려짐**

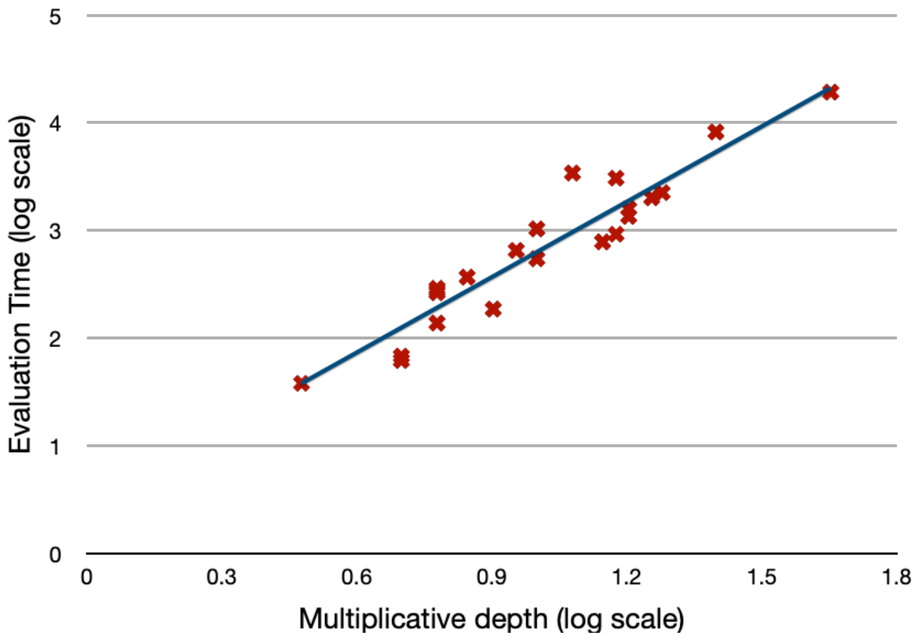
곱셈깊이 - 가장 중요한 성능척도

곱셈깊이 : 입력에서 출력까지 가는 경로를 따라 최대로 수행되는 곱셈(AND) 횟수



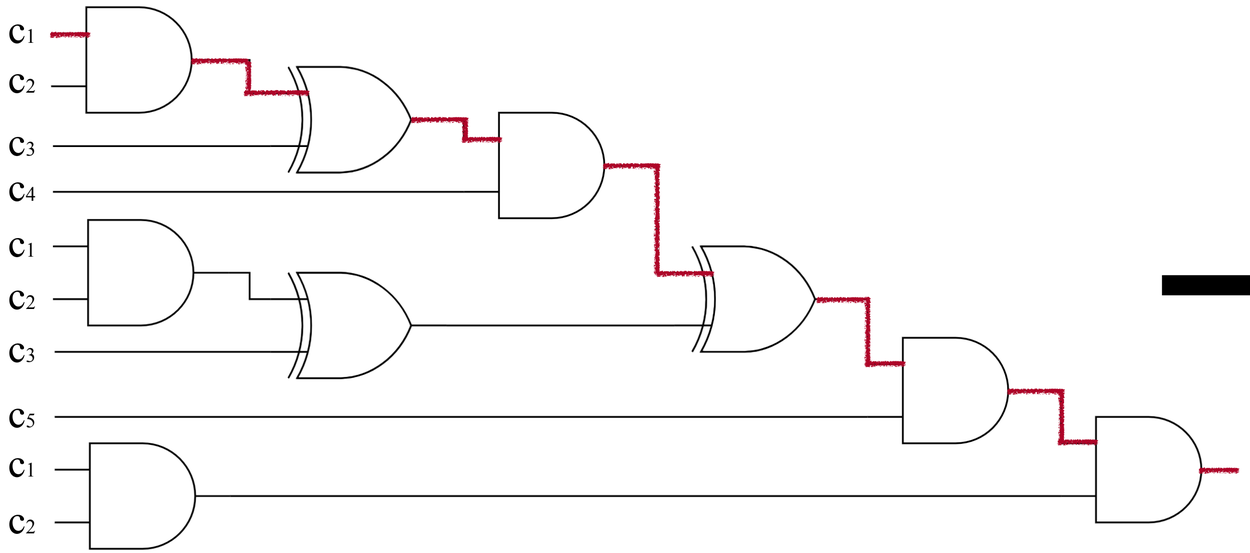
깊이 4



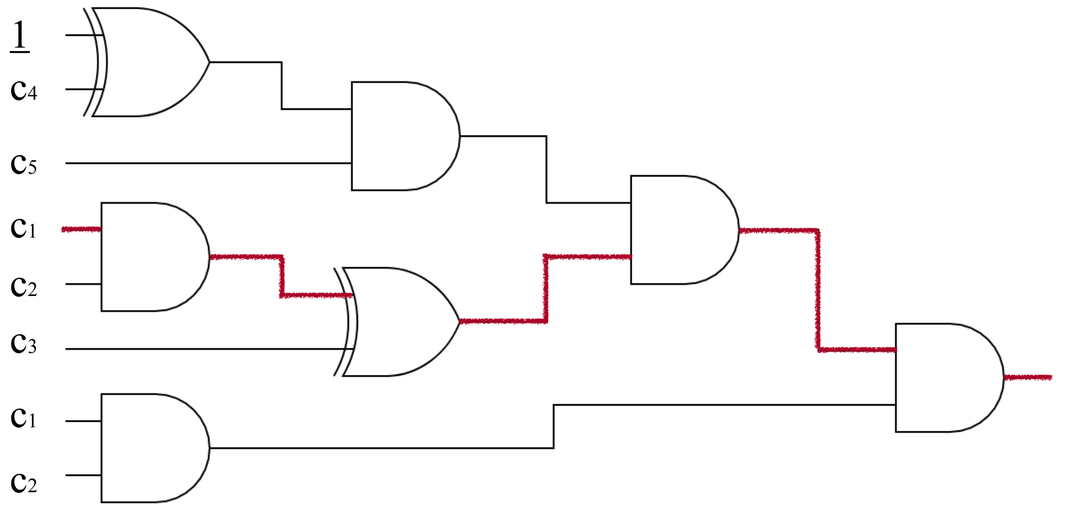


동형암호 최적화란?

- 더 작은 곱셈깊이를 가진 새로운 회로를 찾는것

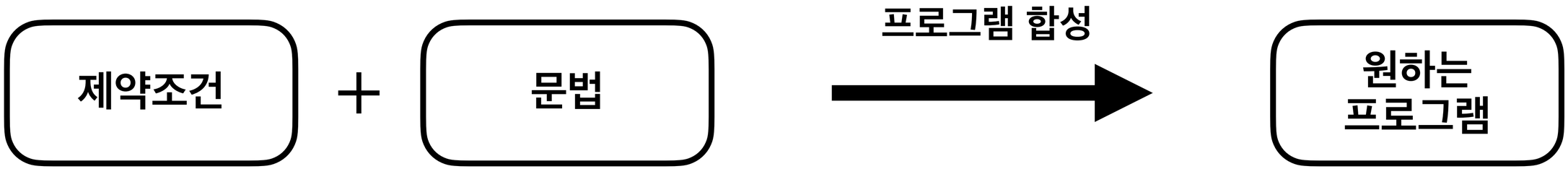


깊이 4



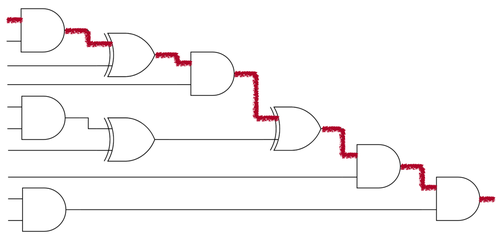
깊이 3

프로그램 합성을 통한 동형암호 회로 최적화



프로그램 합성을 통한 동형암호 회로 최적화

제약조건 + 문법 → (프로그램 합성) → 원하는 프로그램

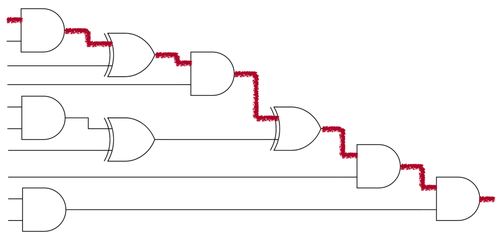


원본과 똑같이 해주세요

(깊이 4)

프로그램 합성을 통한 동형암호 회로 최적화

제약조건 + 문법 → (프로그램 합성) → 원하는 프로그램



원본과 똑같이 해주세요

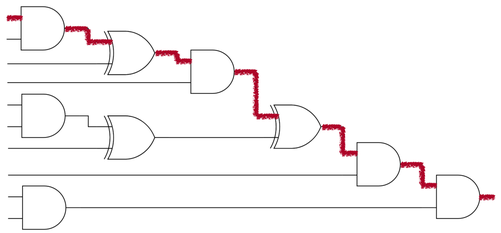
(깊이 4)

$$
\begin{aligned}
S &\rightarrow d_3 \\
d_3 &\rightarrow d_2 \wedge d_2 \mid d_3 \oplus d_3 \mid d_2 \\
d_2 &\rightarrow d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\
d_1 &\rightarrow d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\
d_0 &\rightarrow 0 \mid 1 \mid c_1 \mid c_2 \mid c_3 \mid c_4 \mid c_5
\end{aligned}
$$

깊이는 더 작게 (3 이하)

프로그램 합성을 통한 동형암호 회로 최적화

제약조건 + 문법 → (프로그램 합성) → 원하는 프로그램

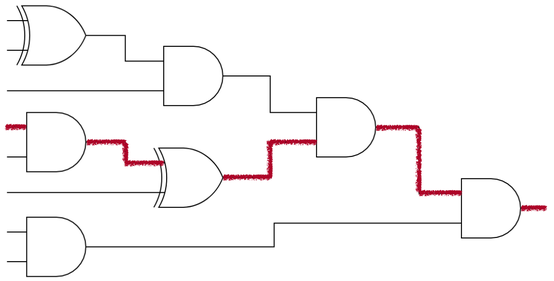


원본과 똑같이 해주세요

(깊이 4)

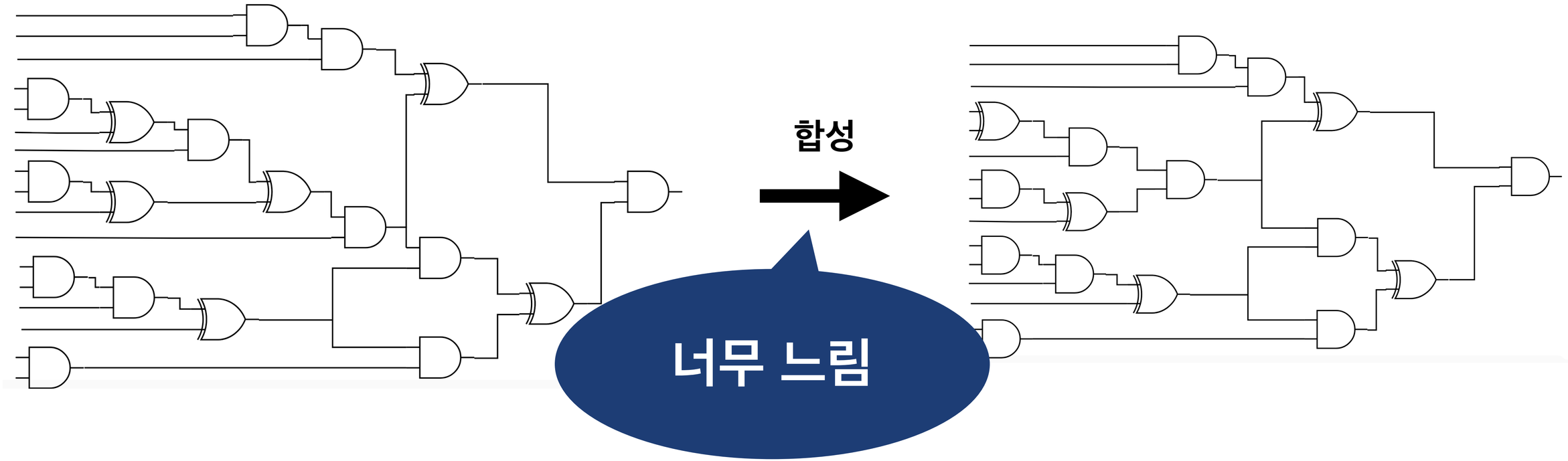
$$
\begin{aligned}
S &\rightarrow d_3 \\
d_3 &\rightarrow d_2 \wedge d_2 \mid d_3 \oplus d_3 \mid d_2 \\
d_2 &\rightarrow d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\
d_1 &\rightarrow d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\
d_0 &\rightarrow 0 \mid 1 \mid c_1 \mid c_2 \mid c_3 \mid c_4 \mid c_5
\end{aligned}
$$

깊이는 더 작게 (3이하)

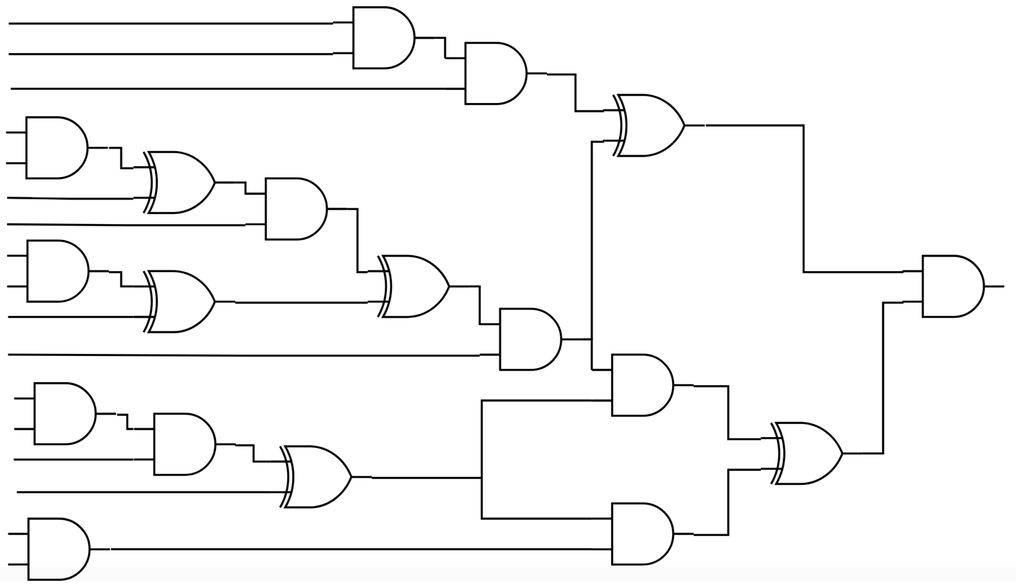


최적화된 회로 (깊이 3)

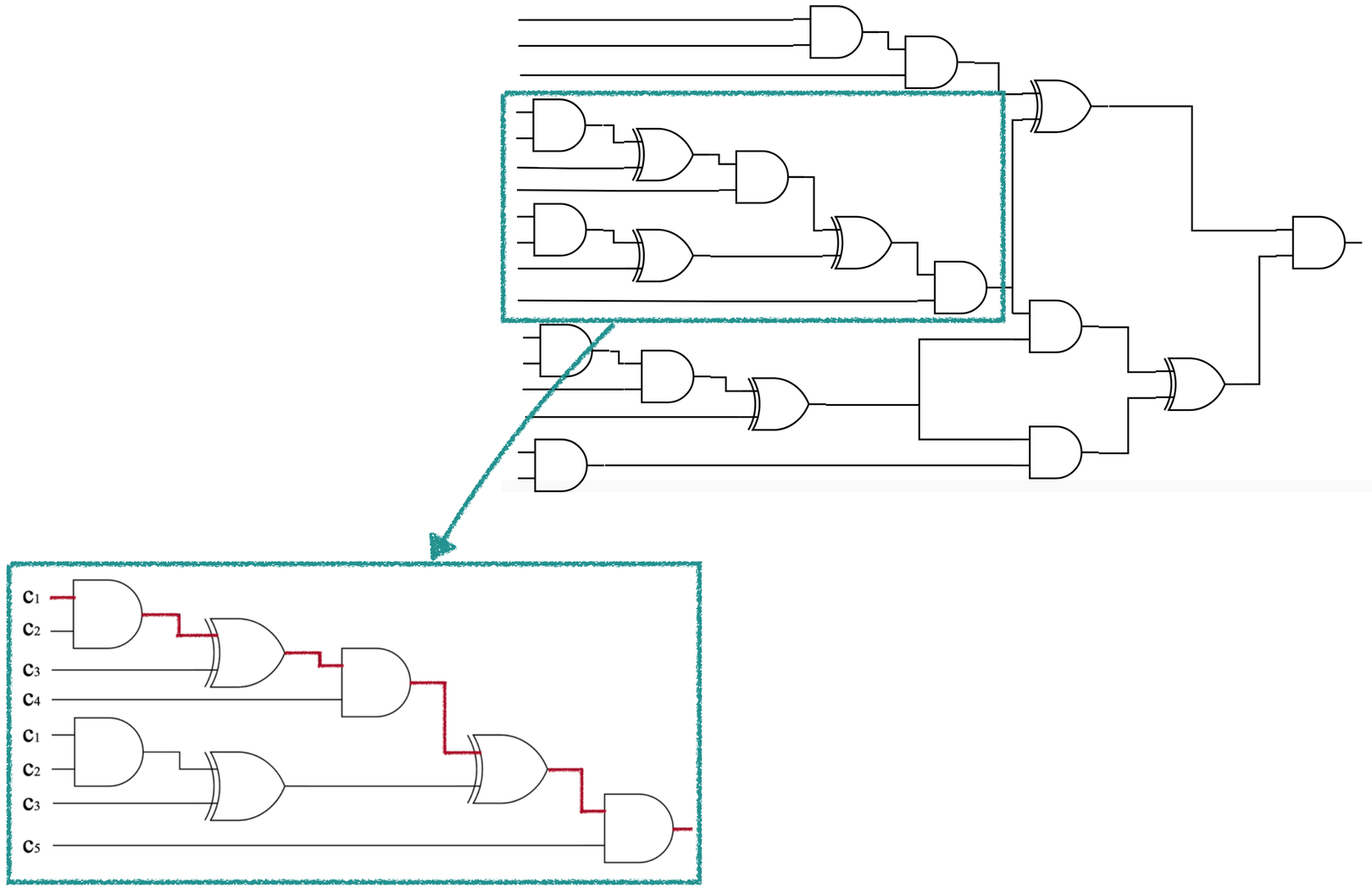
한계점 : 합성기 성능 *Scalability*



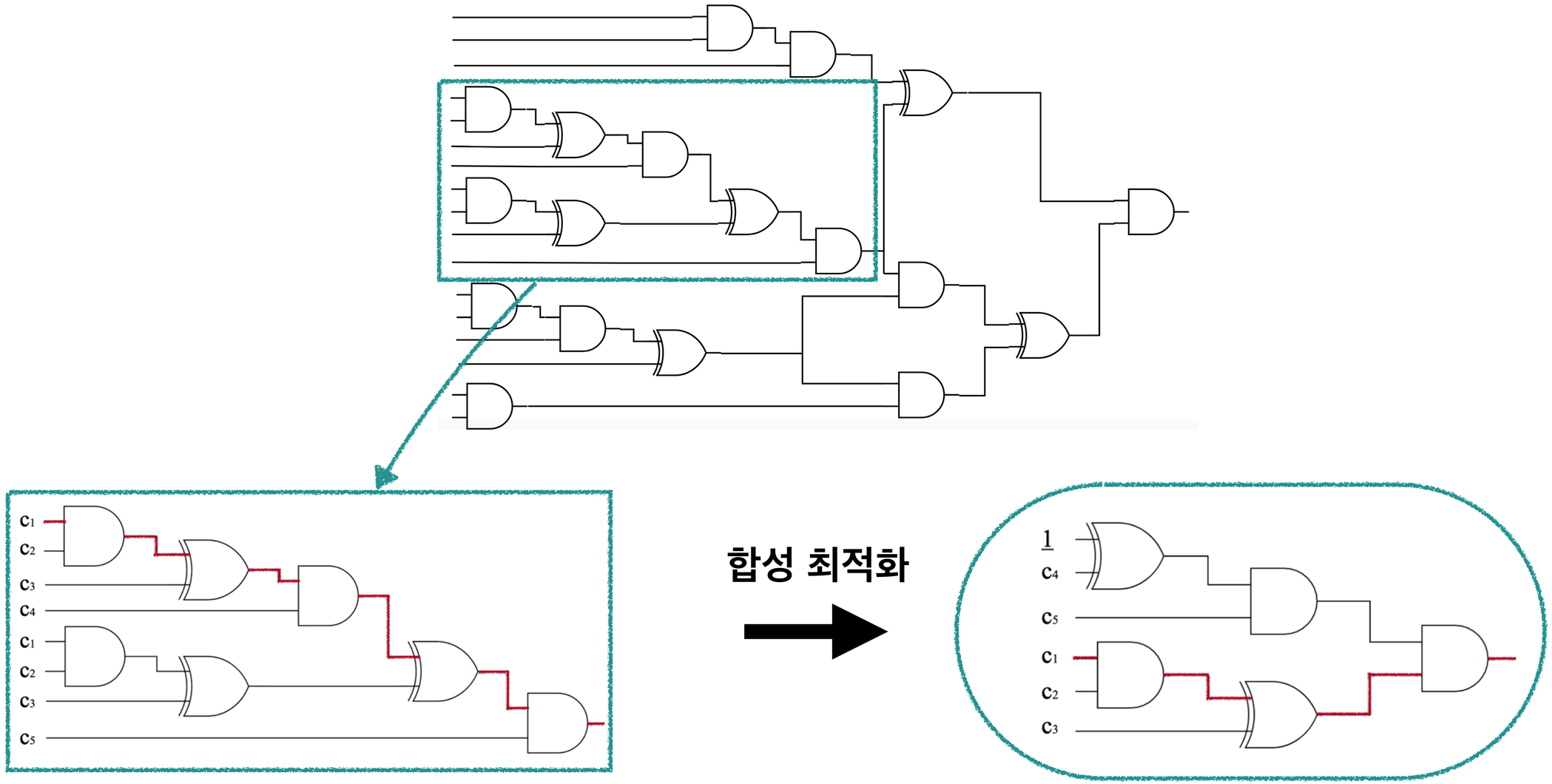
해결책 1 : 부분적으로 합성하기 *Divide-and-Conquer*



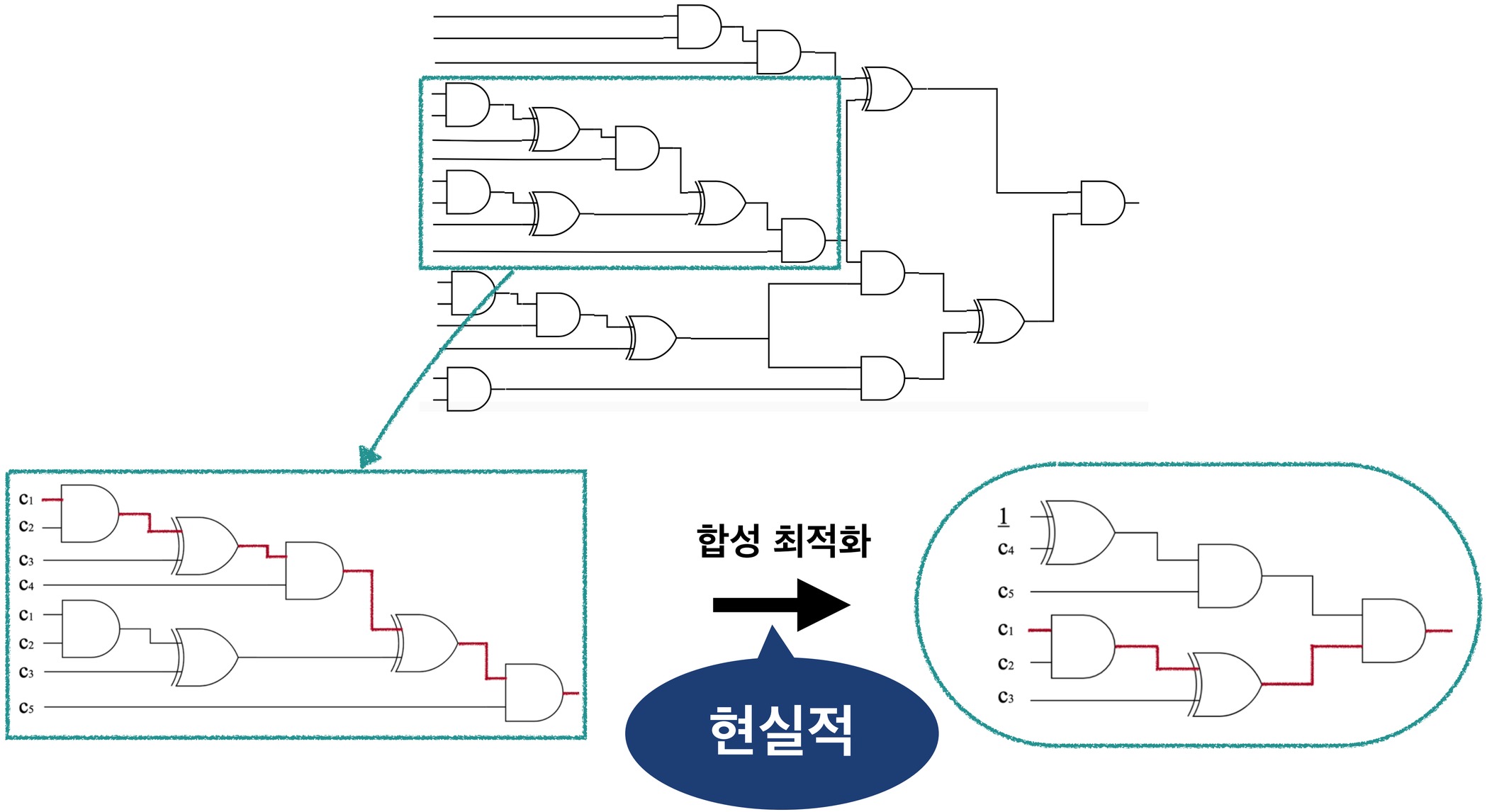
해결책 1 : 부분적으로 합성하기 *Divide-and-Conquer*



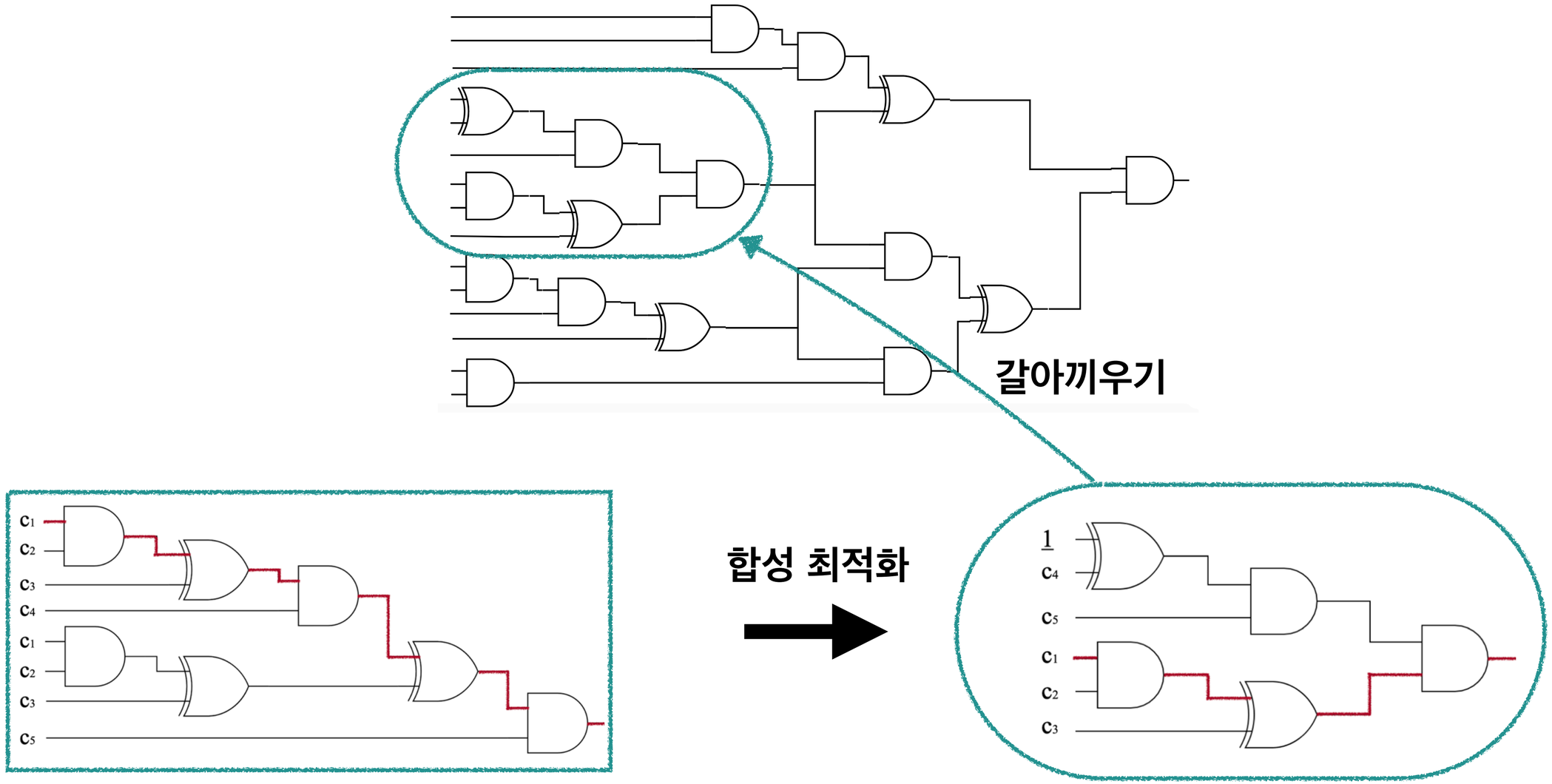
해결책 1 : 부분적으로 합성하기 *Divide-and-Conquer*



해결책 1 : 부분적으로 합성하기 *Divide-and-Conquer*



해결책 1 : 부분적으로 합성하기 *Divide-and-Conquer*



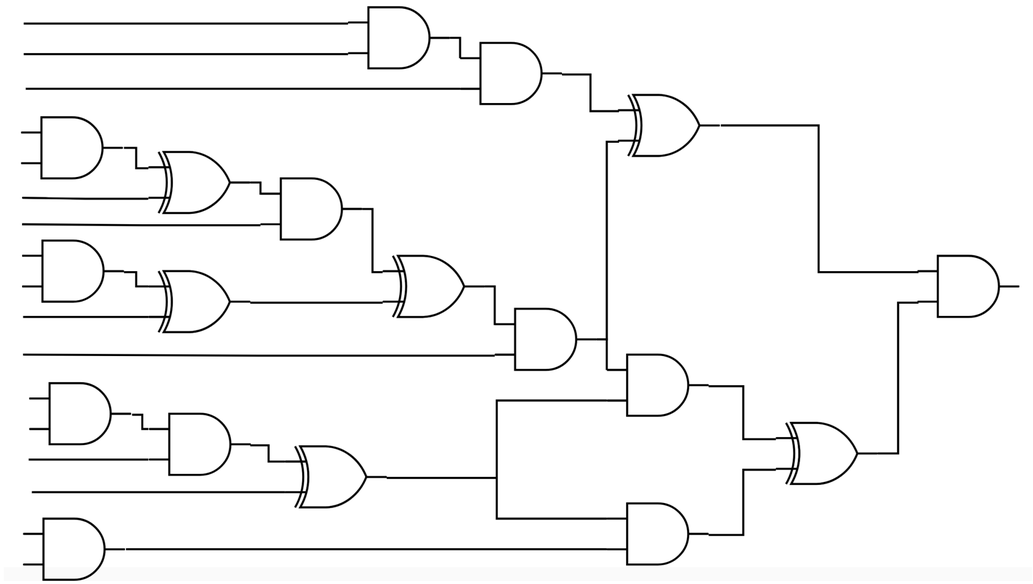
해결책 2 : 성공적인 최적화 패턴 기억하기

- 규칙 학습 단계
  - 합성 최적화에 성공한 패턴들을 최적화 규칙으로 학습

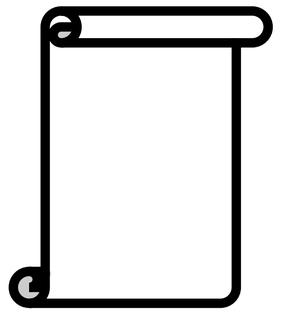
- 규칙 적용 단계
  - 식 다시쓰기 기술로 학습한 패턴 적용하기

합성 최적화 패턴 수집단계

학습할 동형암호
프로그램

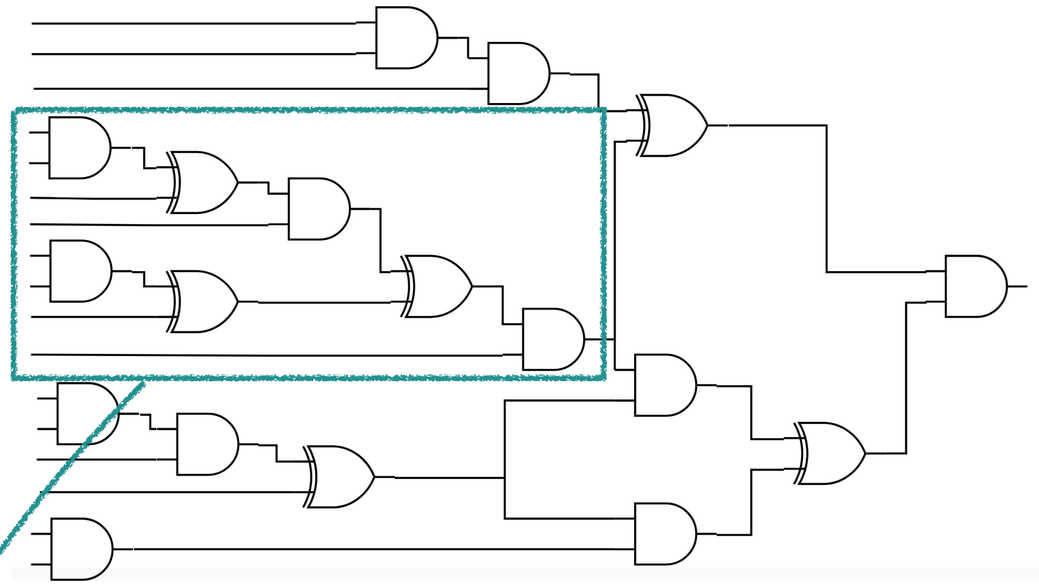


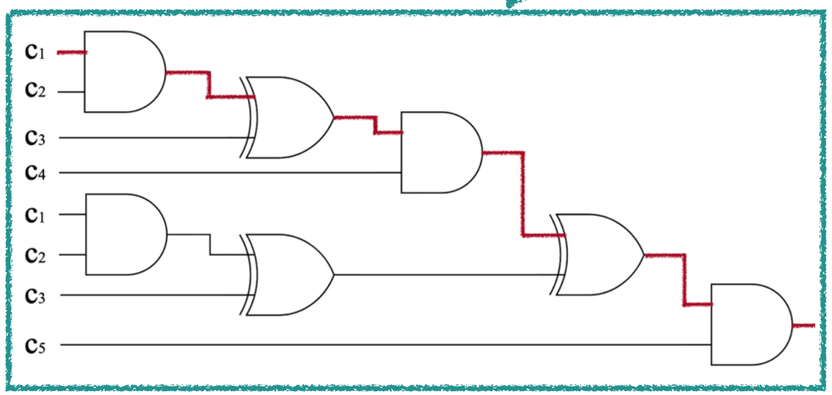
수집한
최적화 패턴들



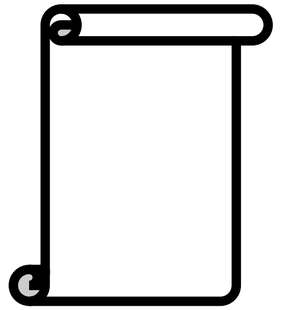
합성 최적화 패턴 수집단계

학습할 동형암호
프로그램



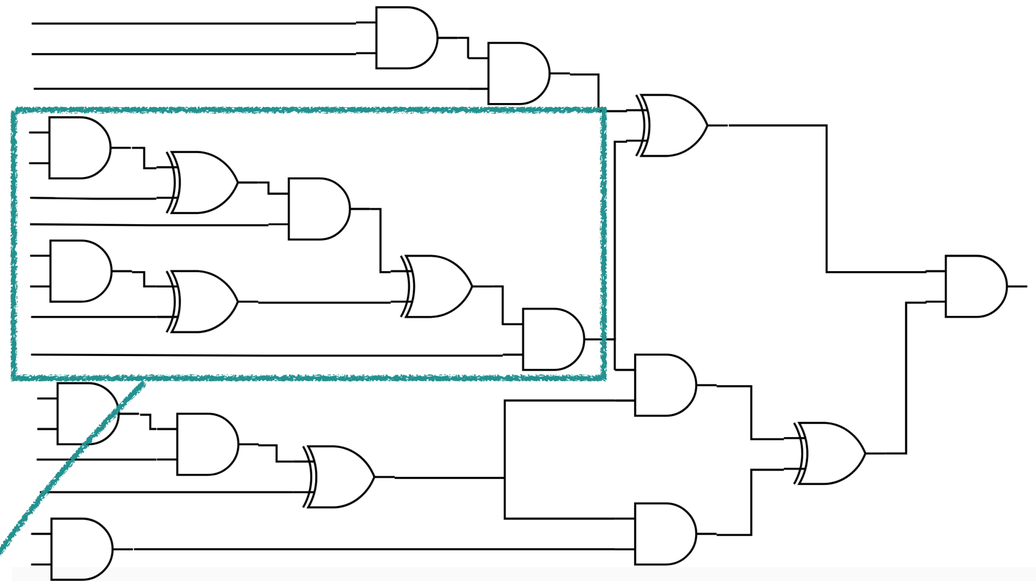


수집한
최적화 패턴들

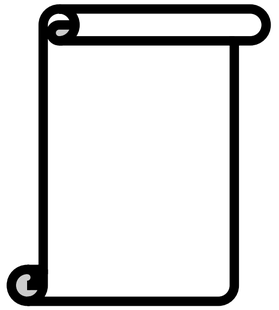


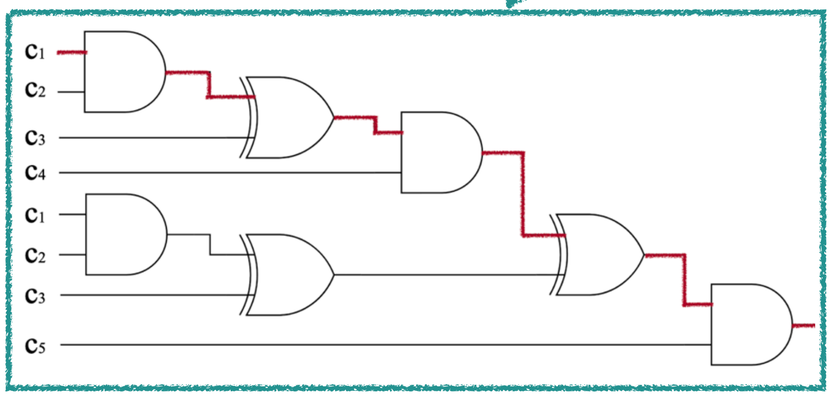
합성 최적화 패턴 수집단계

학습할 동형암호
프로그램

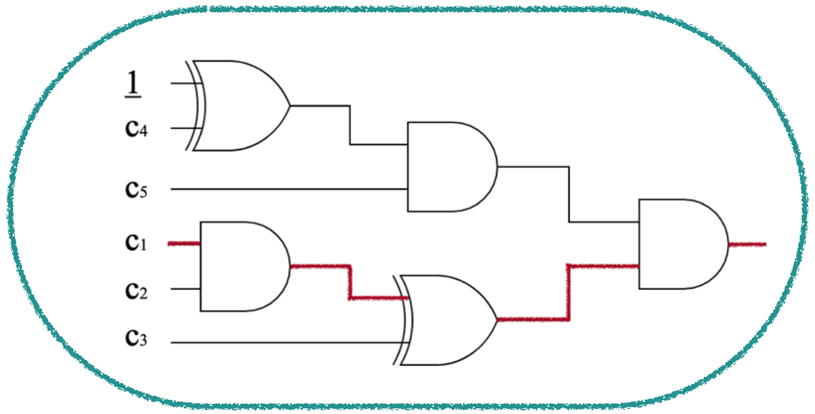


수집한
최적화 패턴들



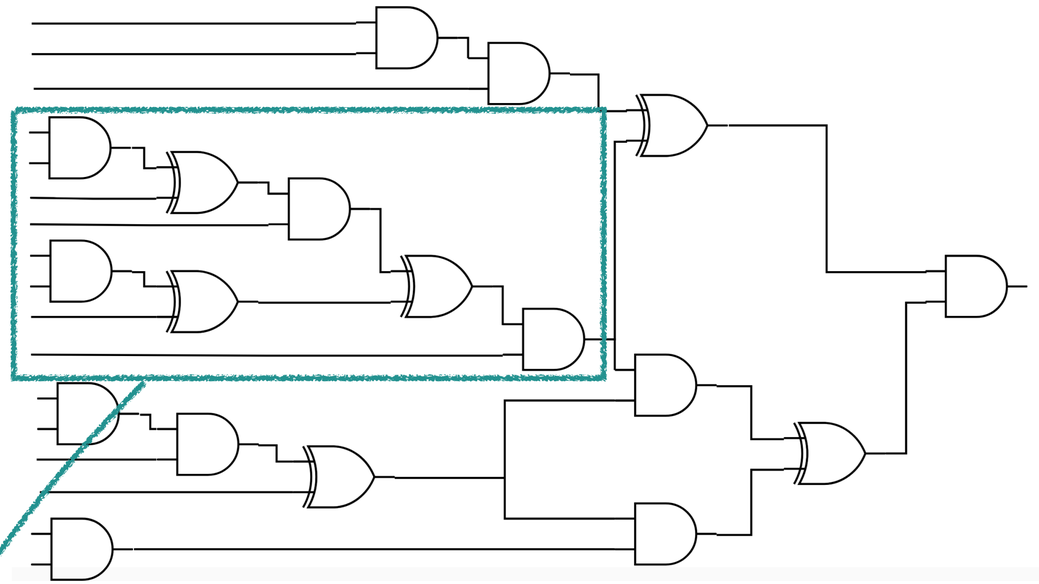


합성 최적화

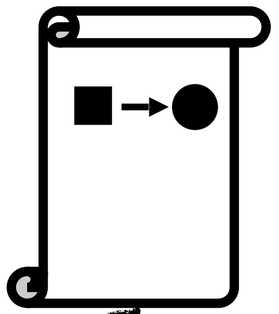


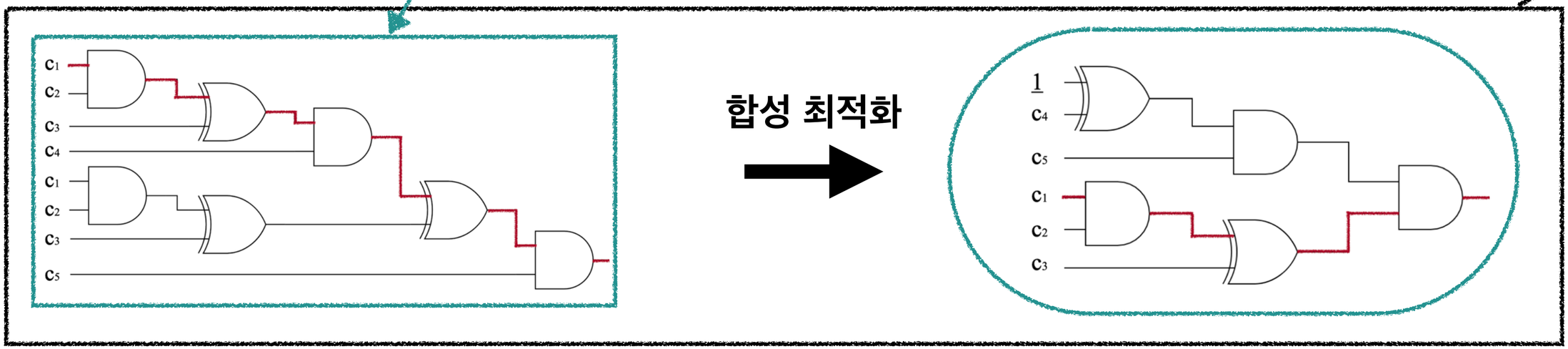
합성 최적화 패턴 수집단계

학습할 동형암호
프로그램



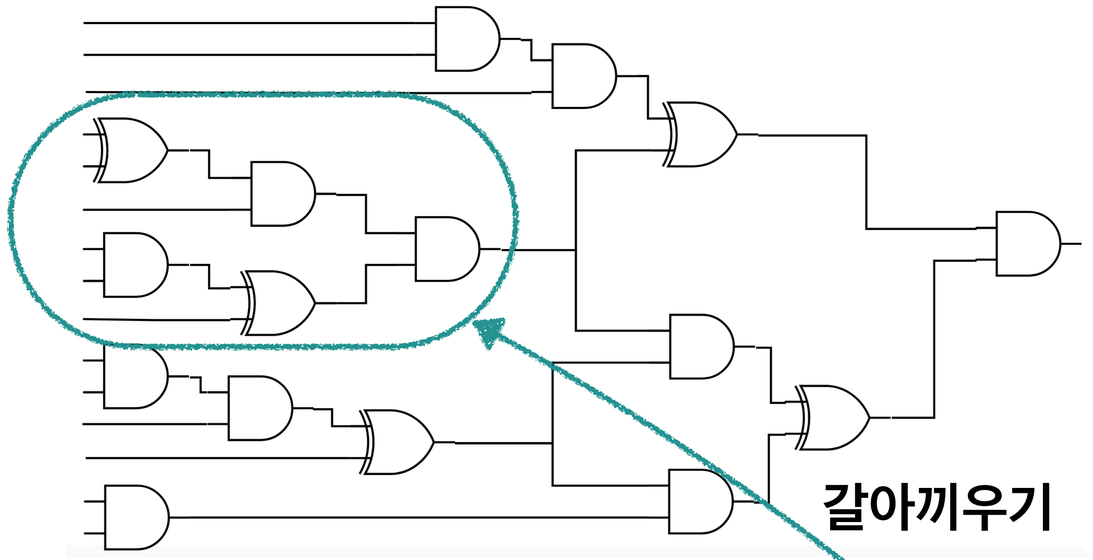
수집한
최적화 패턴들





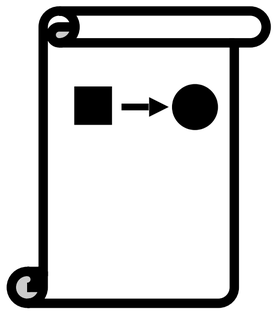
합성 최적화 패턴 수집단계

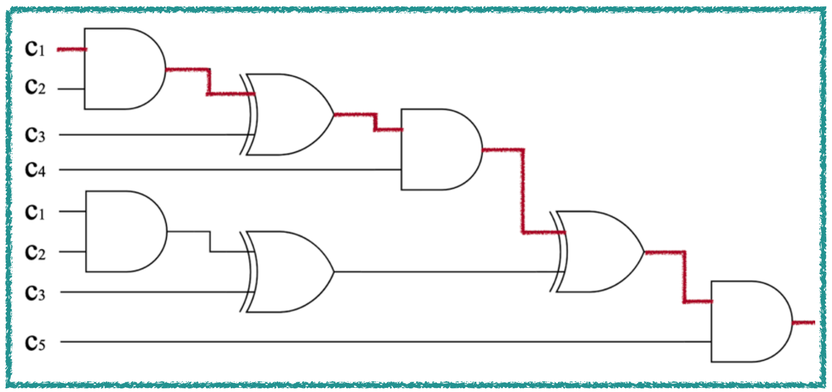
학습할 동형암호
프로그램



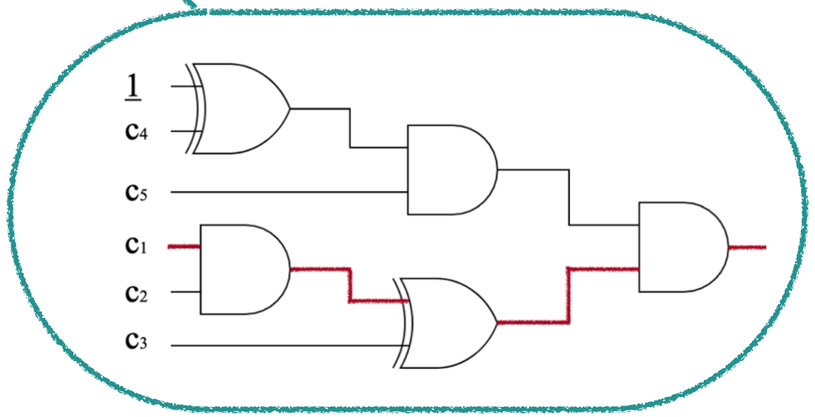
갈아끼우기

수집한
최적화 패턴들



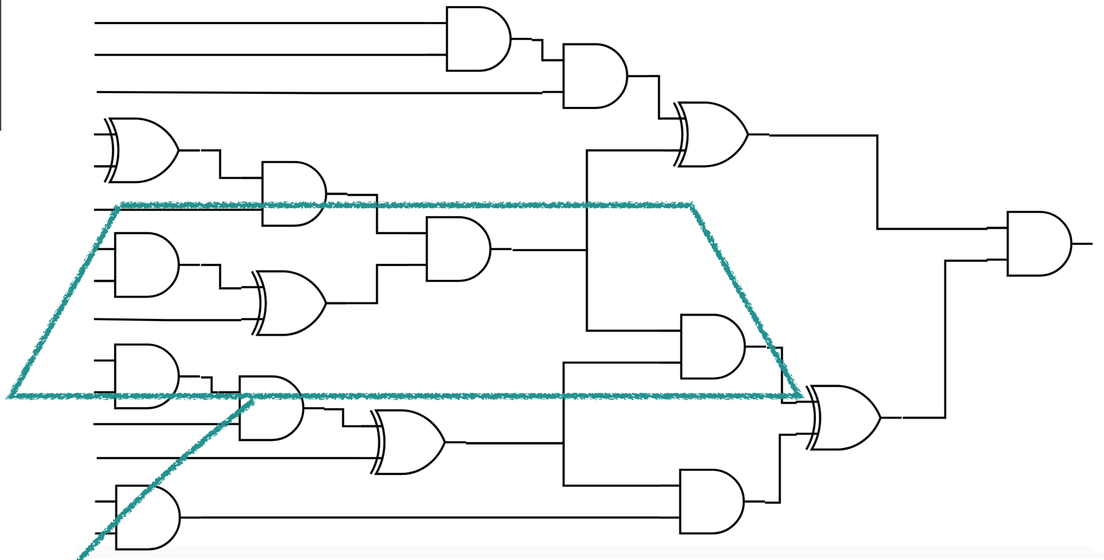


합성 최적화

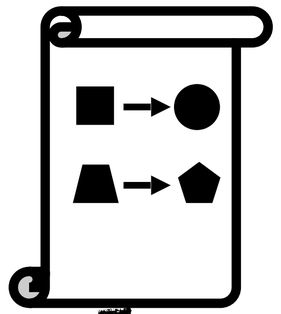


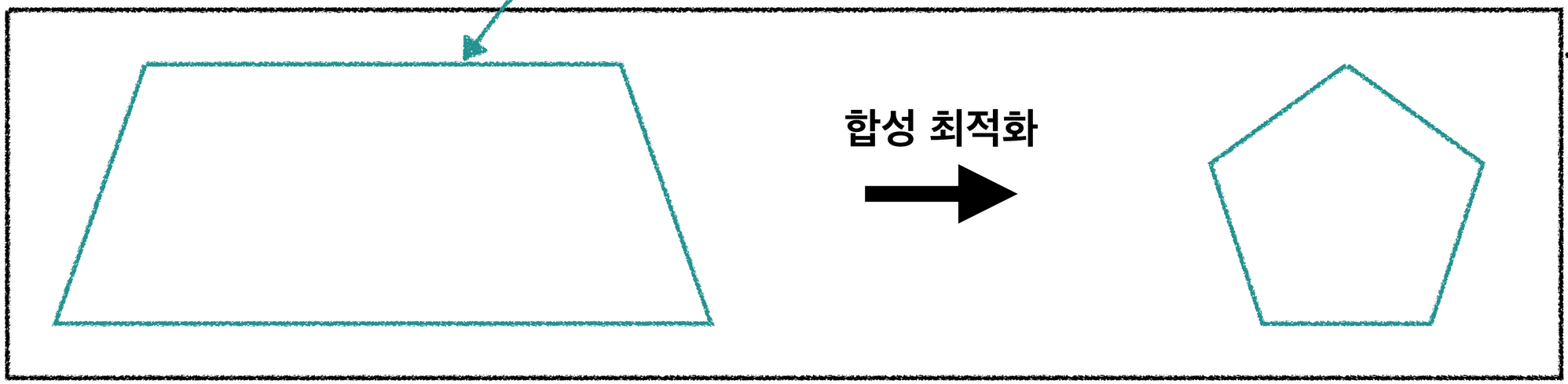
합성 최적화 패턴 수집단계

학습할 동형암호
프로그램



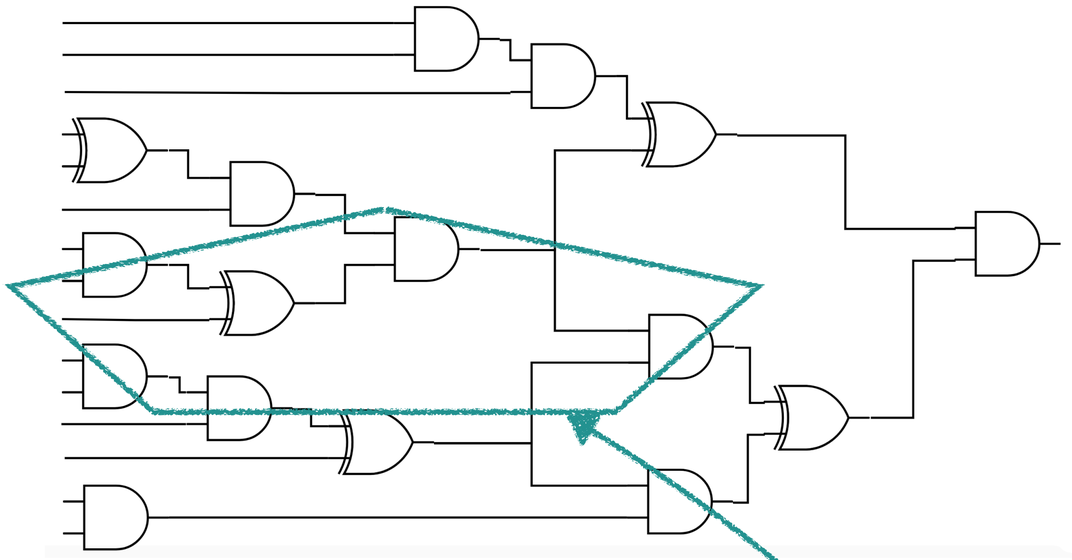
수집한
최적화 패턴들



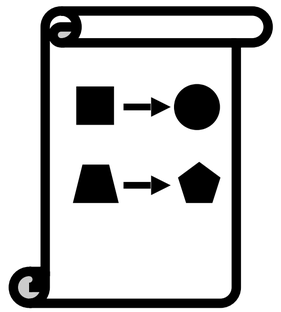


합성 최적화 패턴 수집단계

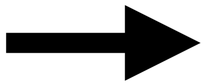
학습할 동형암호
프로그램

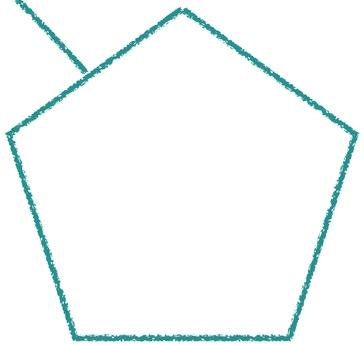


수집한
최적화 패턴들



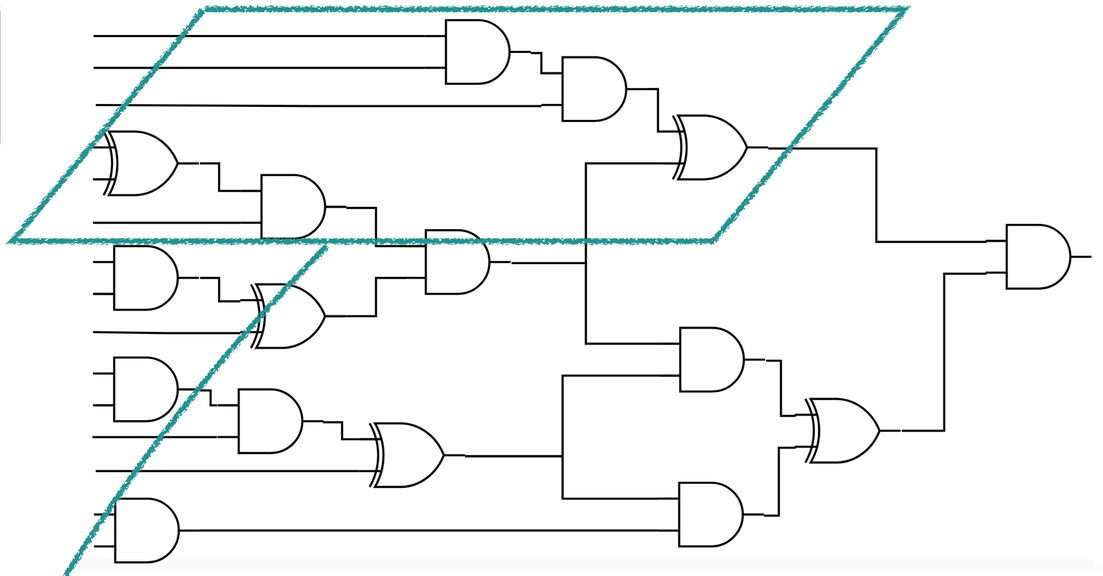
합성 최적화



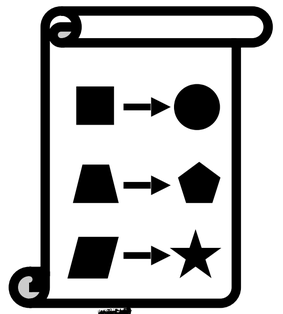


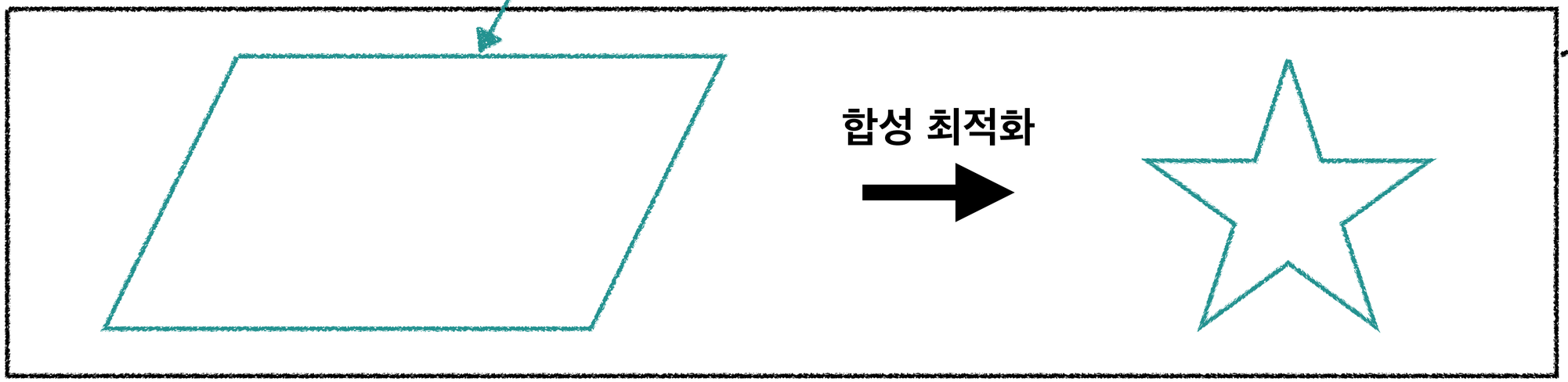
합성 최적화 패턴 수집단계

학습할 동형암호
프로그램



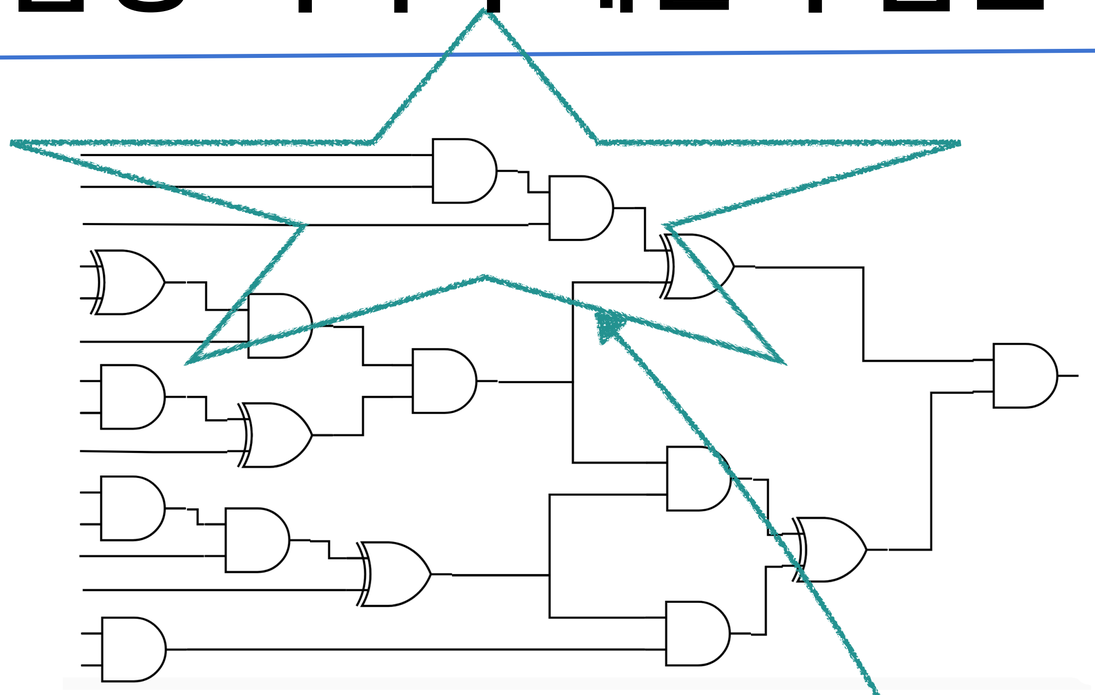
수집한
최적화 패턴들



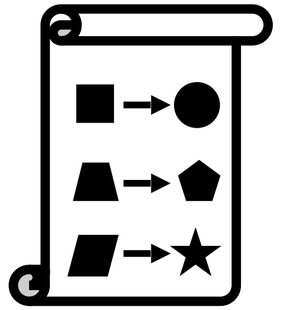


합성 최적화 패턴 수집단계

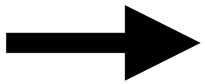
학습할 동형암호
프로그램



수집한
최적화 패턴들

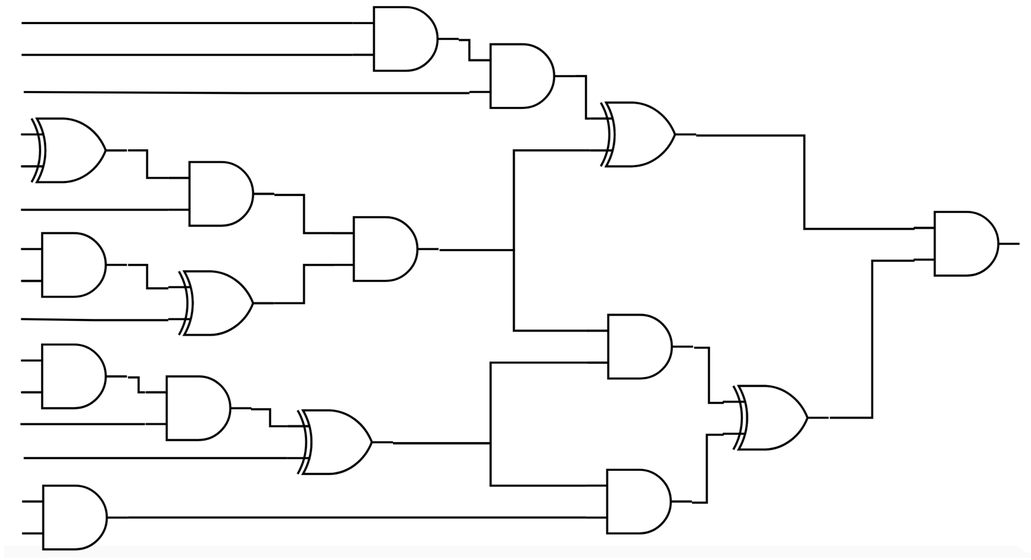


합성 최적화

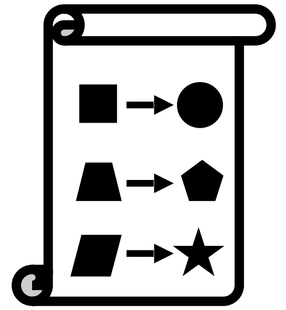


합성 최적화 패턴 수집단계

학습할 동형암호
프로그램

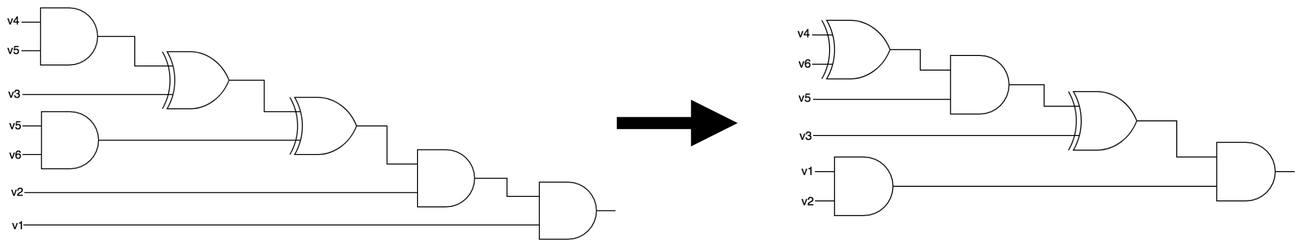


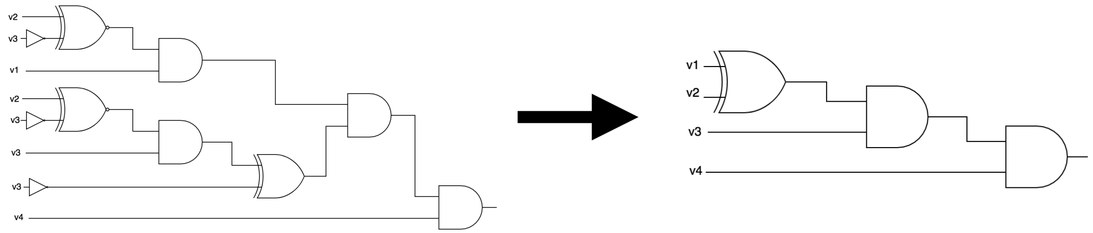
수집한
최적화 패턴들

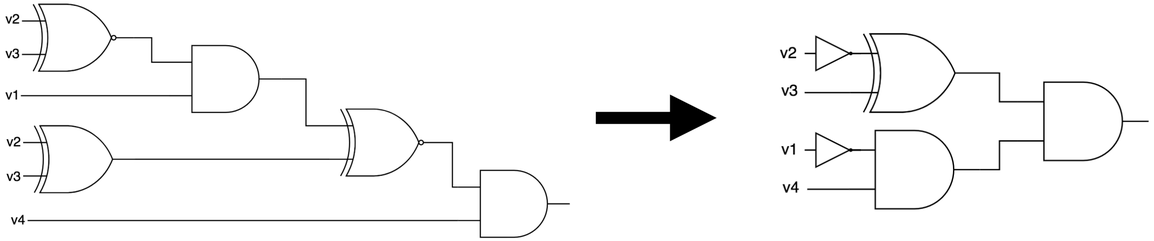


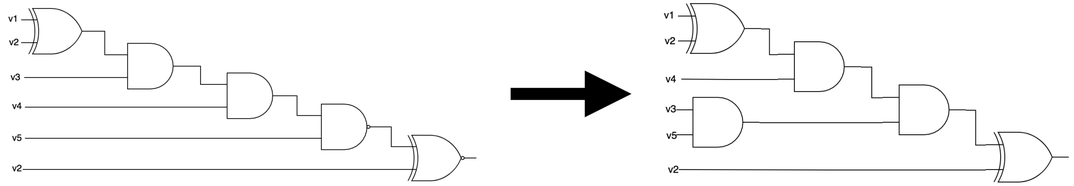
400여가지 규칙 발견

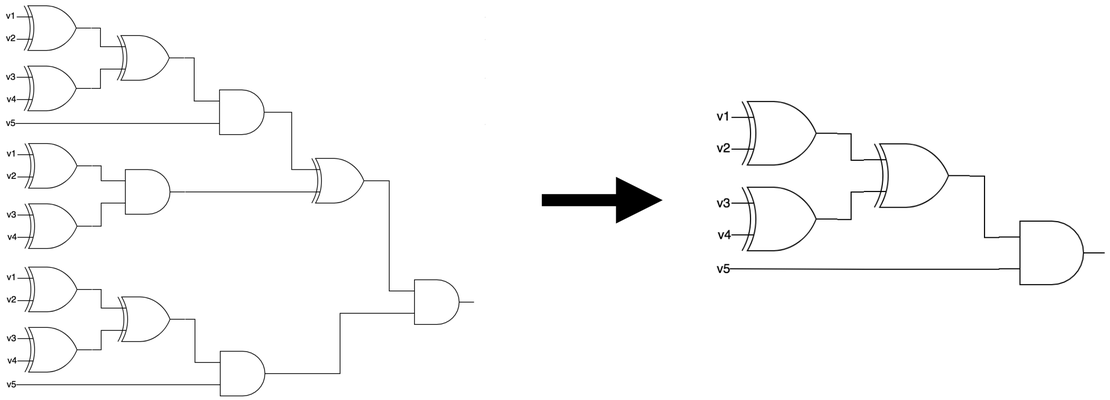
수집된 최적화 합성 패턴

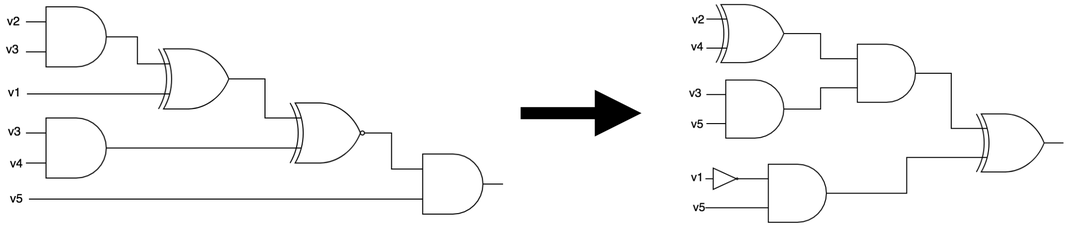

v4
v5
v3
v5
v6
v2
v1
v4
v6
v5
v3
v1
v2


v2
v3
v1
v2
v3
v3
v3
v4
v1
v2
v3
v4


v2
v3
v1
v2
v3
v4
v2
v3
v1
v4

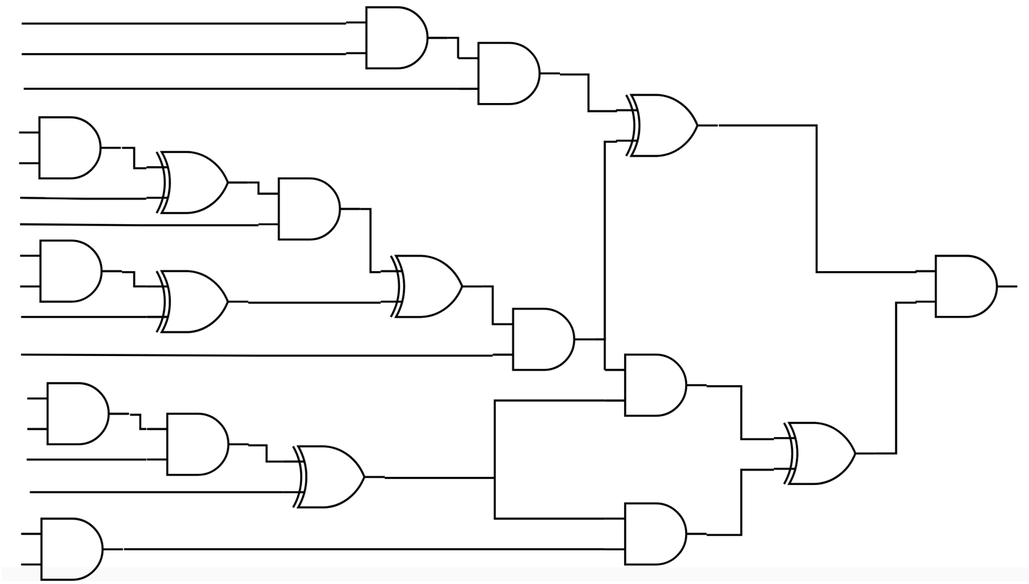

v1
v2
v3
v4
v5
v2
v1
v2
v4
v3
v5
v2


v1
v2
v3
v4
v5
v1
v2
v3
v4
v1
v2
v3
v4
v5
v1
v2
v3
v4
v5

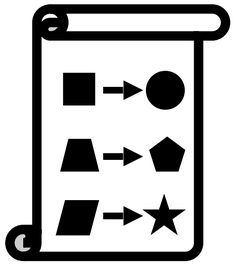

v2
v3
v1
v3
v4
v5
v2
v4
v3
v5
v1
v5

최적화 규칙 적용단계

입력 동형암호
프로그램

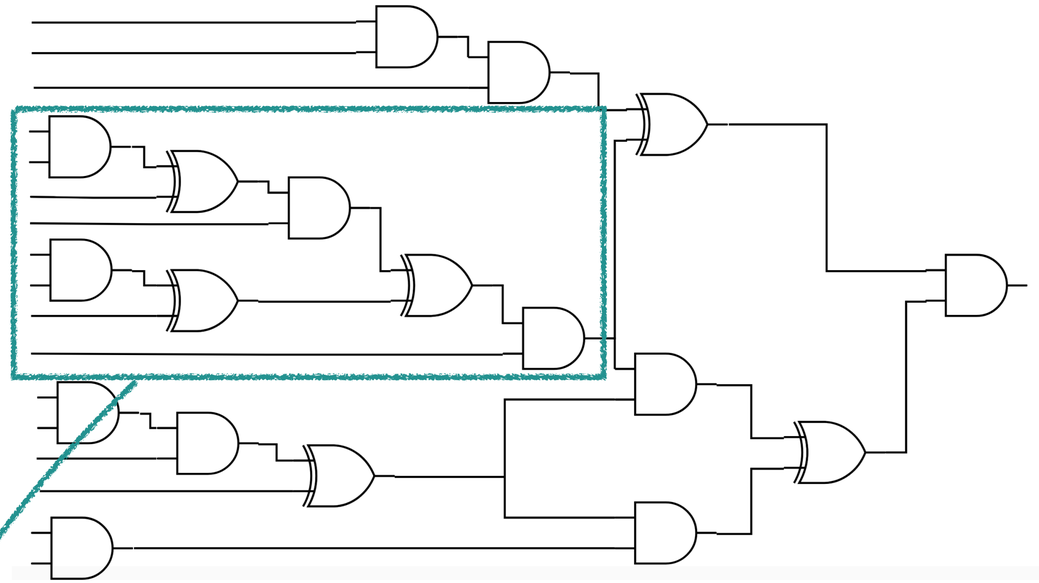


수집한
최적화 패턴들

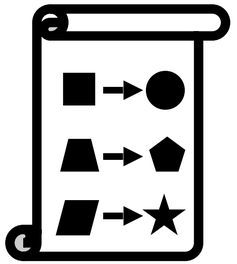


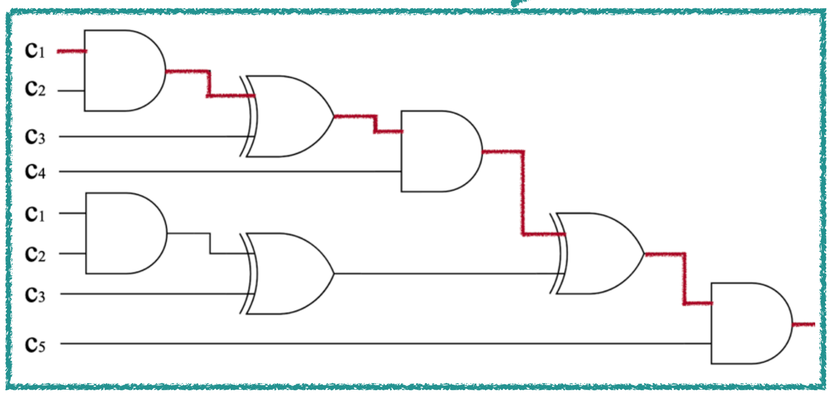
최적화 규칙 적용단계

입력 동형암호
프로그램

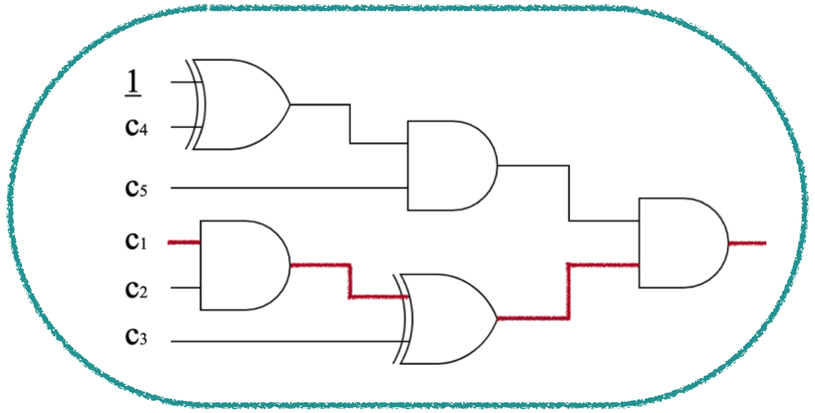


수집한
최적화 패턴들



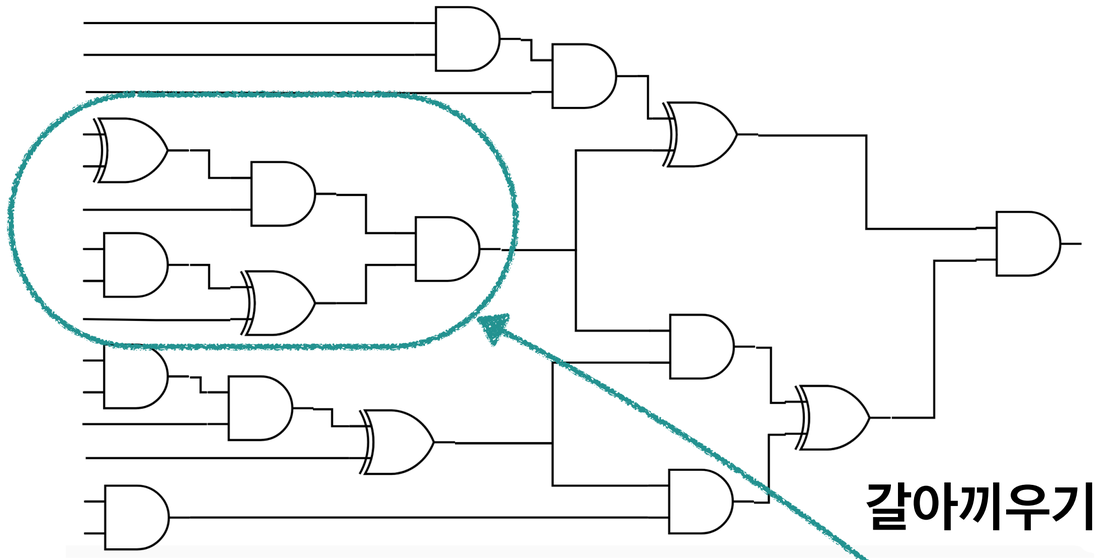


최적화 규칙
적용

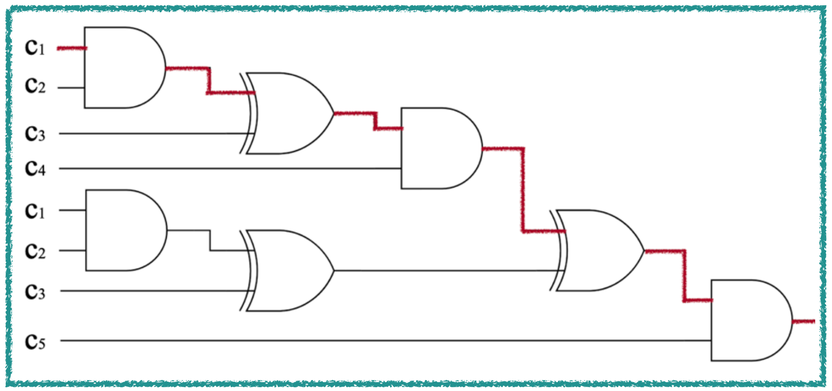


최적화 규칙 적용단계

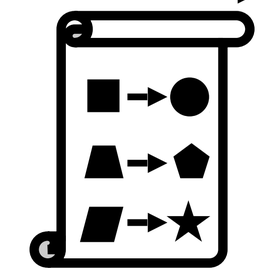
입력 동형암호
프로그램

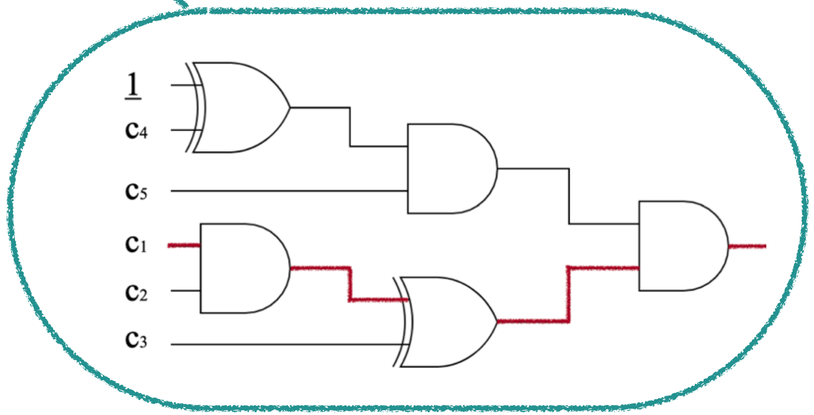


갈아끼우기



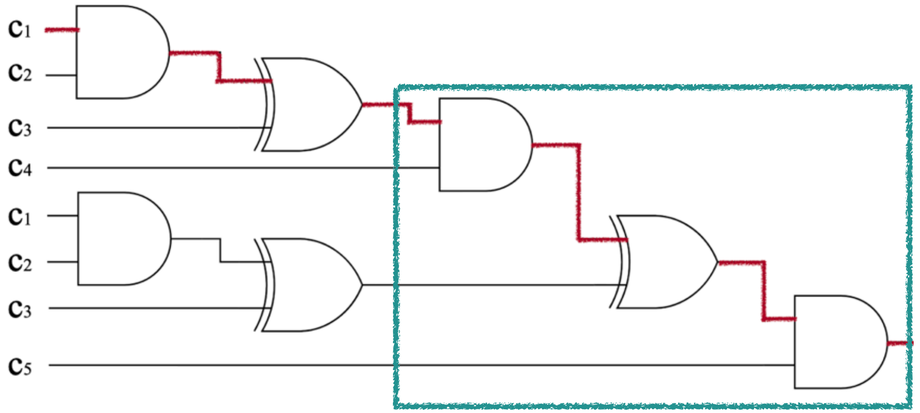
최적화 규칙
적용



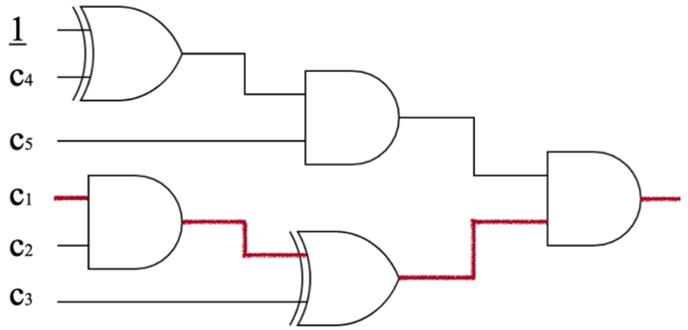


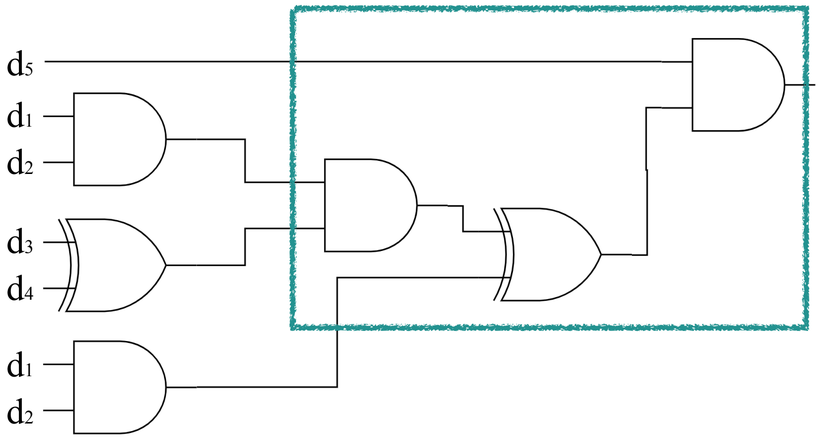
최적화 규칙 유연하게 적용하기 (1/2)

단순 모양비교로는 명확한 한계



학습한
최적화 규칙



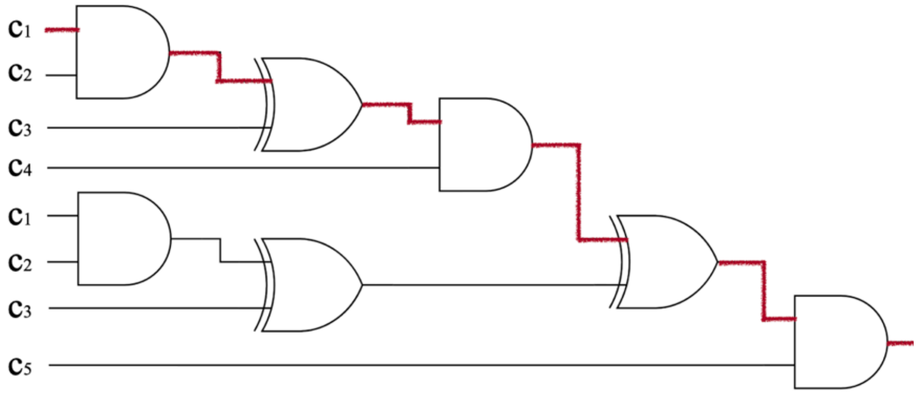


입력 회로
최적화

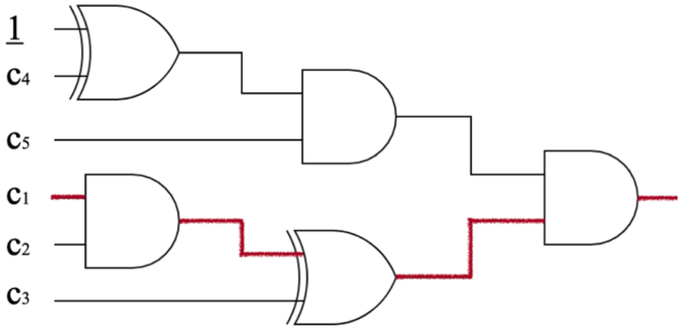
?

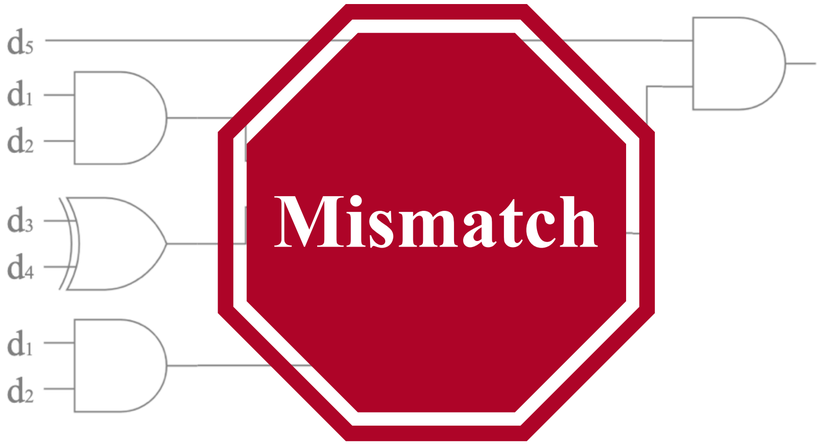
최적화 규칙 유연하게 적용하기 (1/2)

단순 모양비교로는 명확한 한계



학습한
최적화 규칙



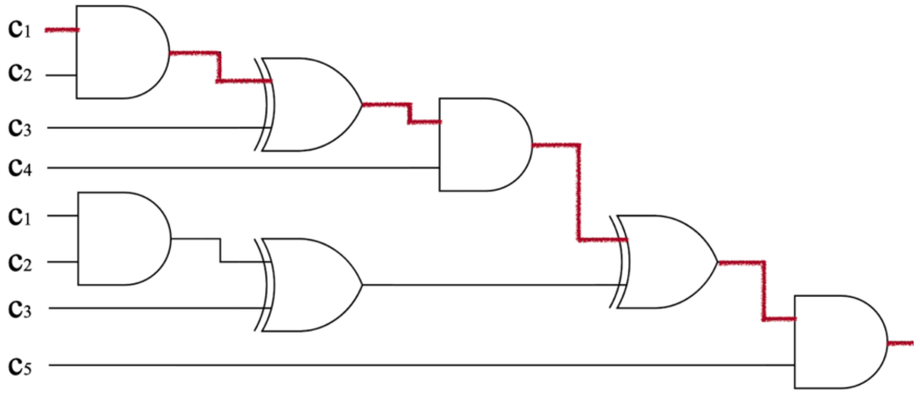


입력 회로
최적화

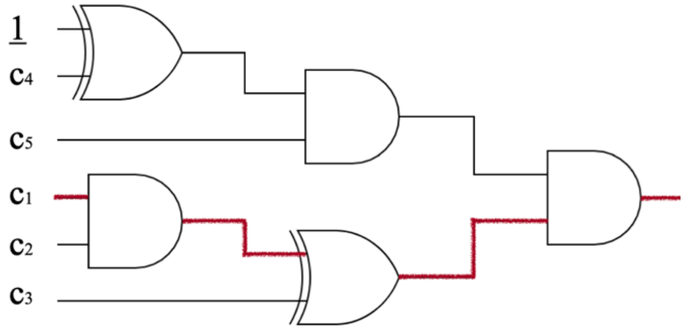
?

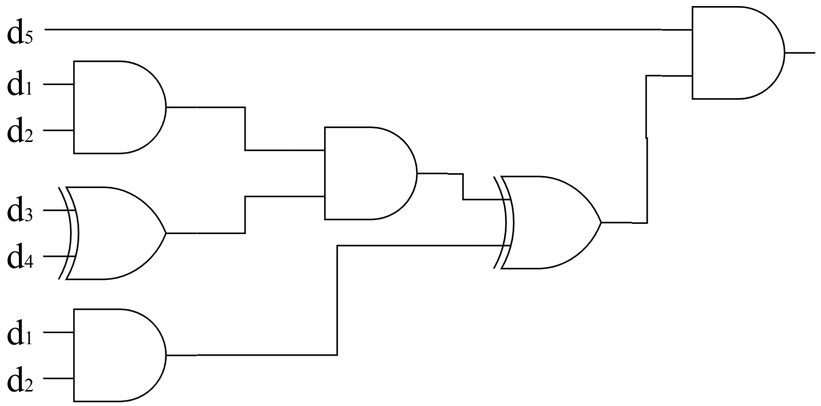
최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



학습한
최적화 규칙



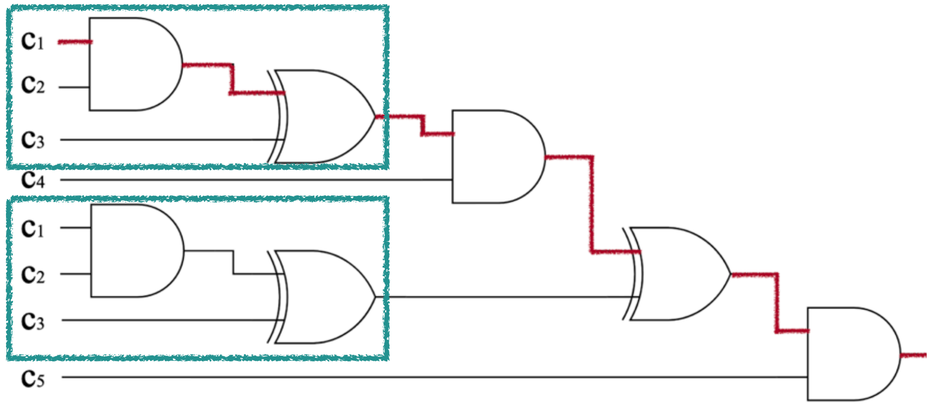


입력 회로
최적화

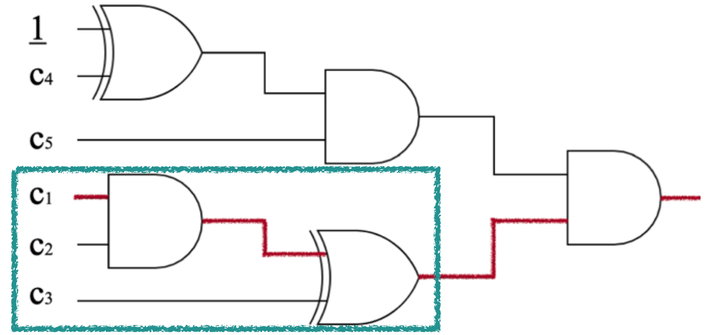
?

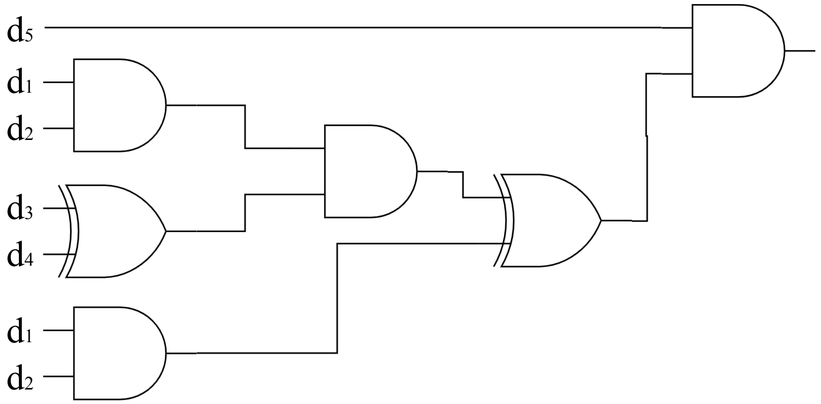
최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



학습한
최적화 규칙



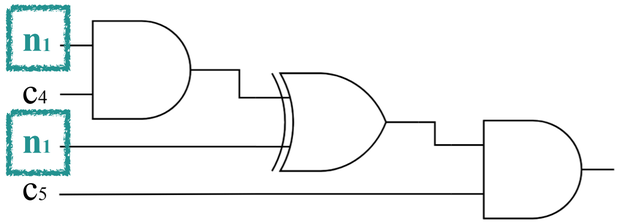


입력 회로
최적화

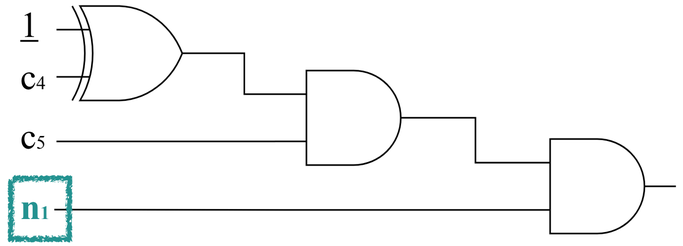
?

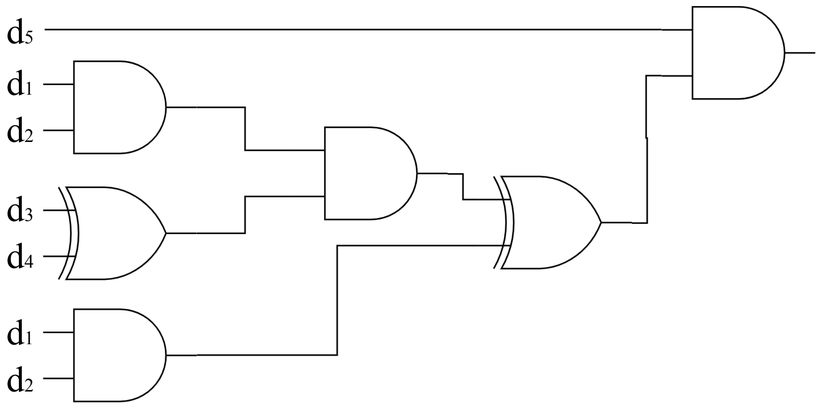
최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



학습한
최적화 규칙



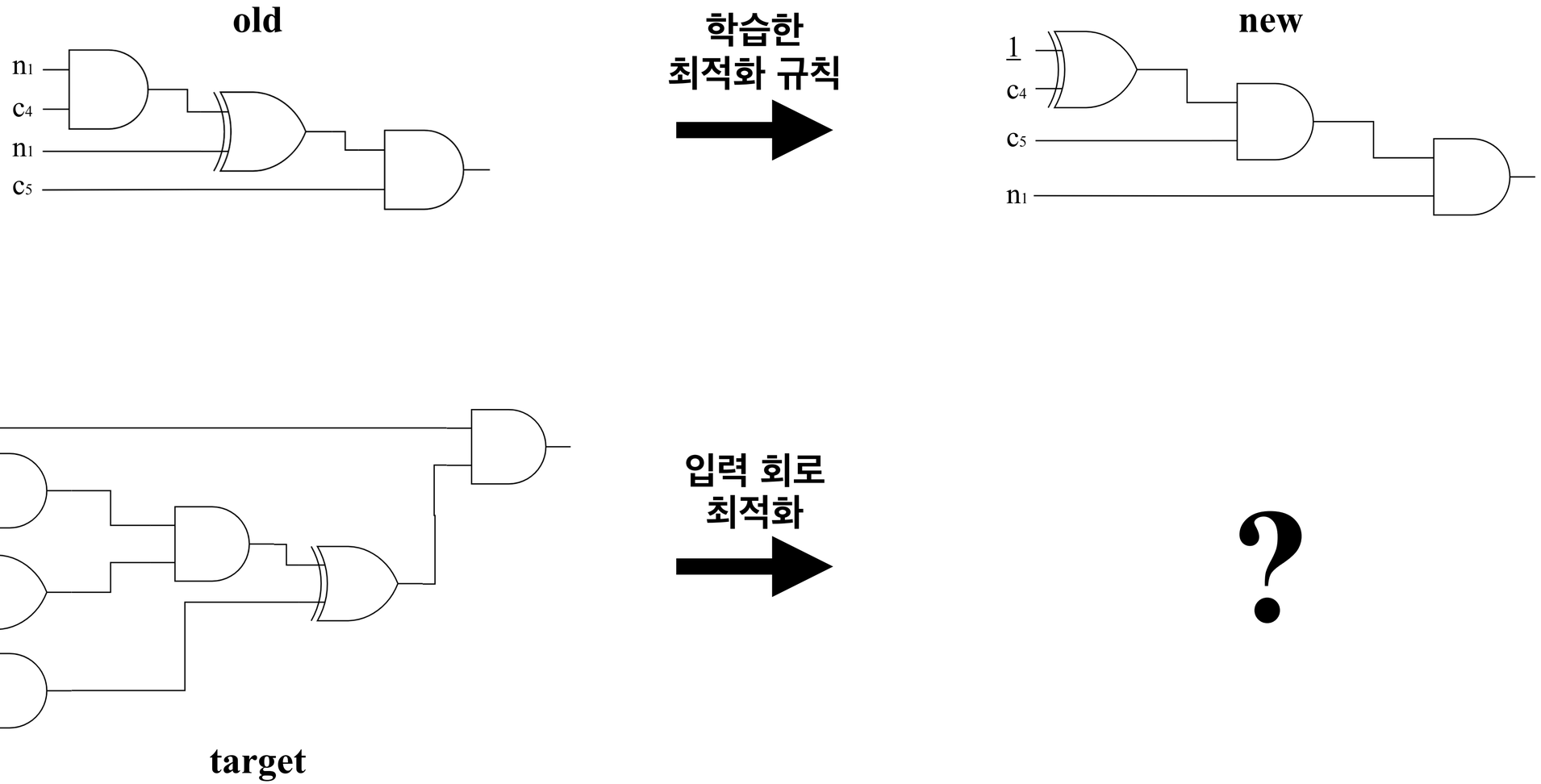


입력 회로
최적화

?

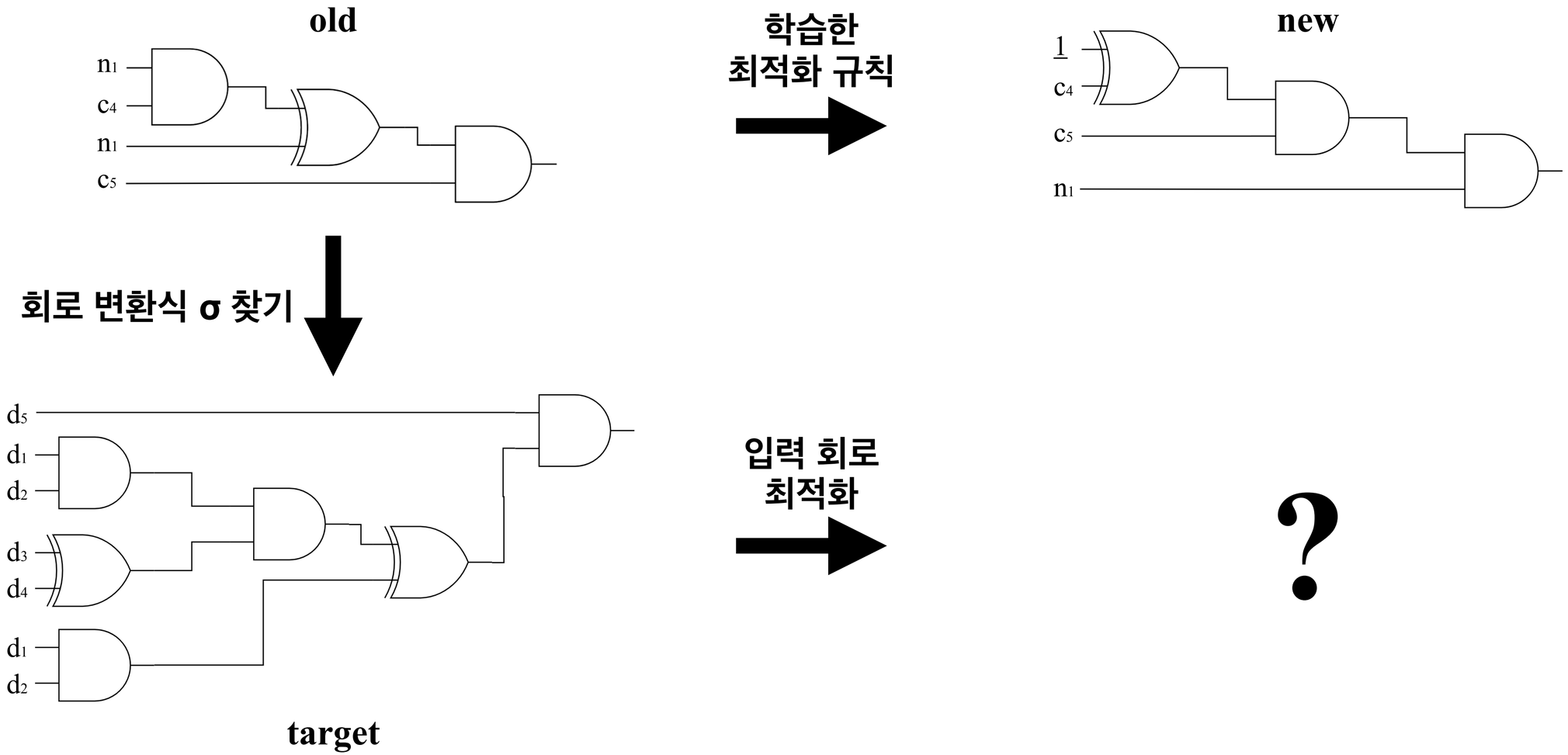
최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



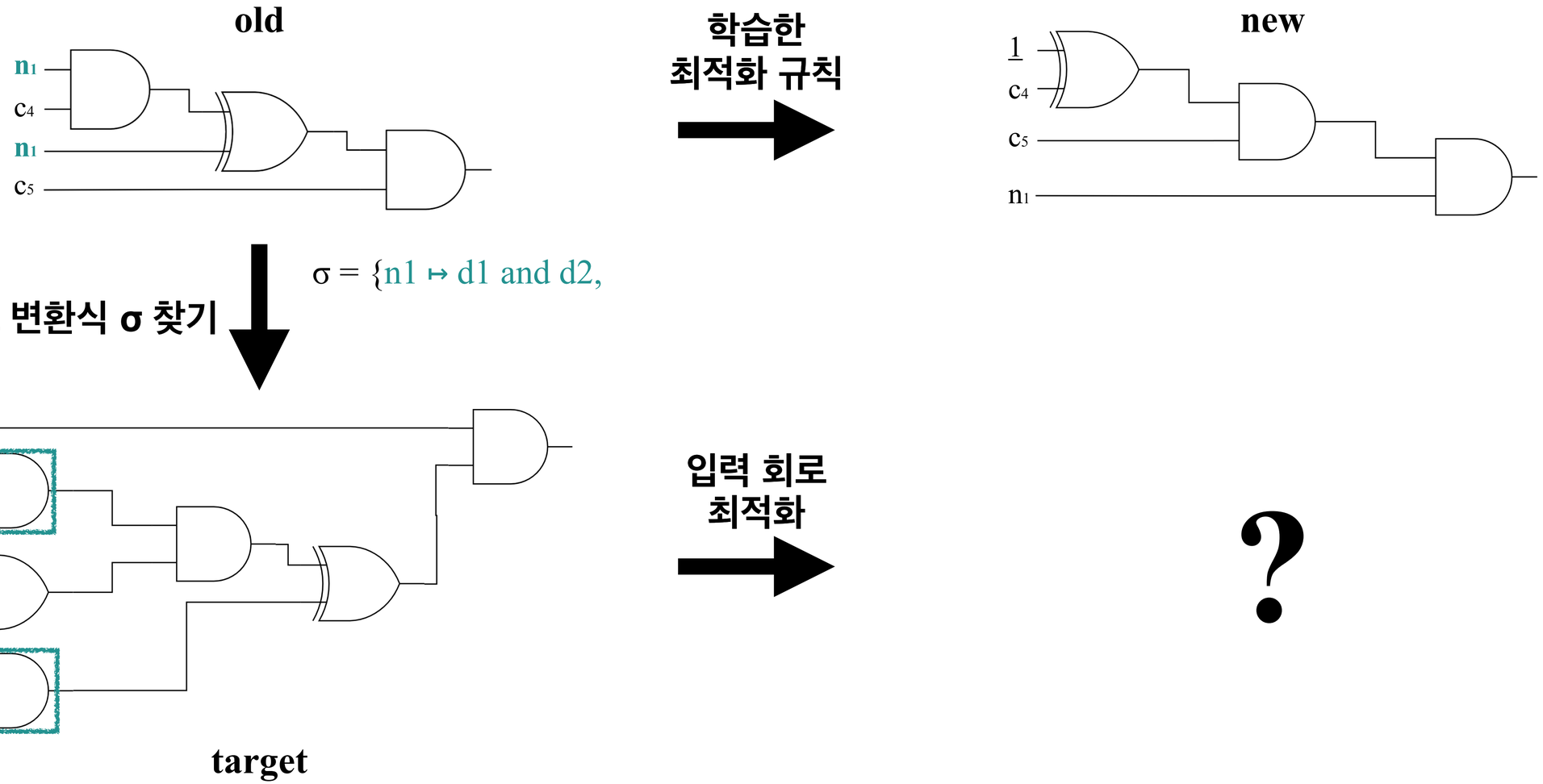
최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



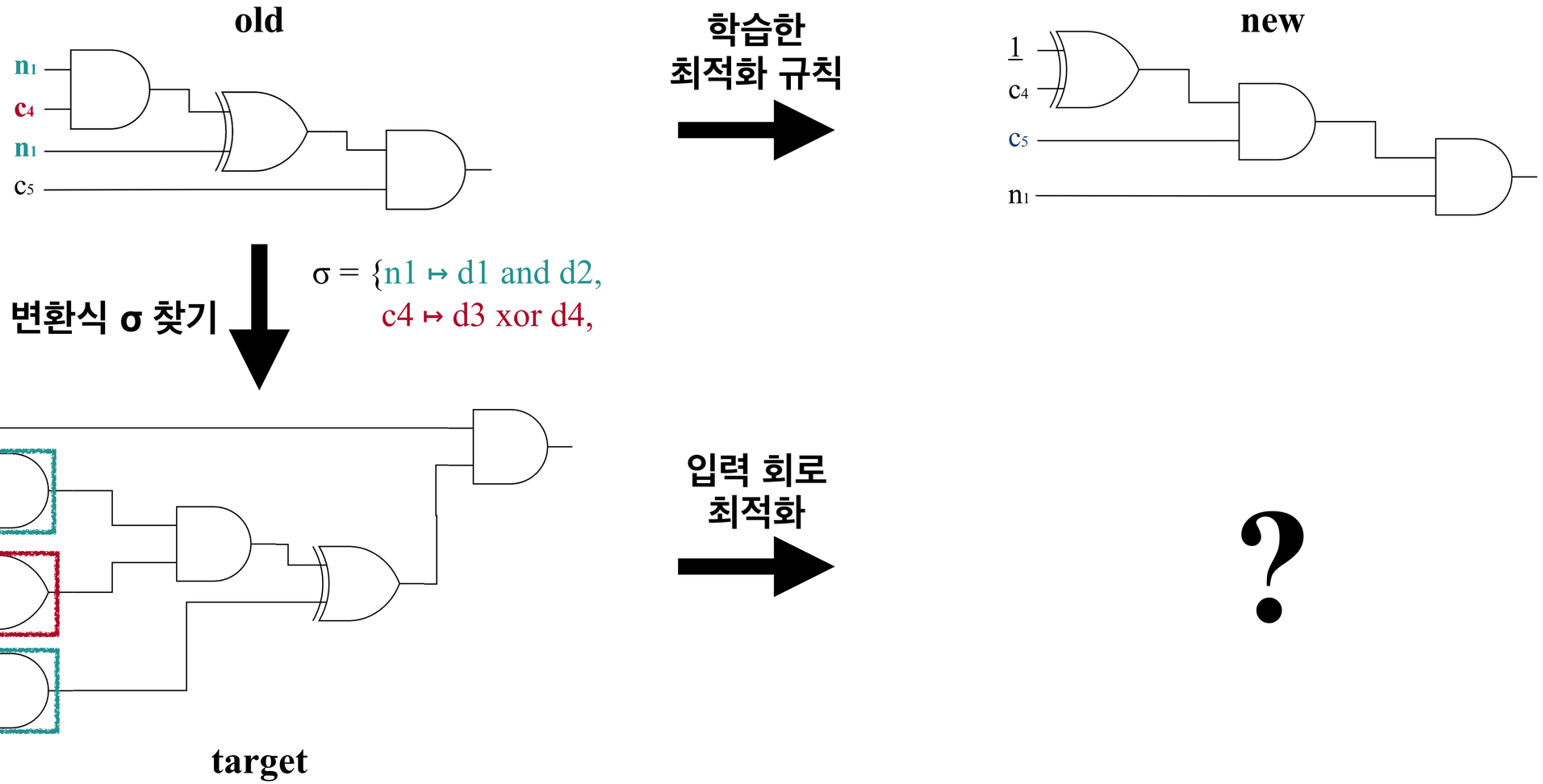
최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



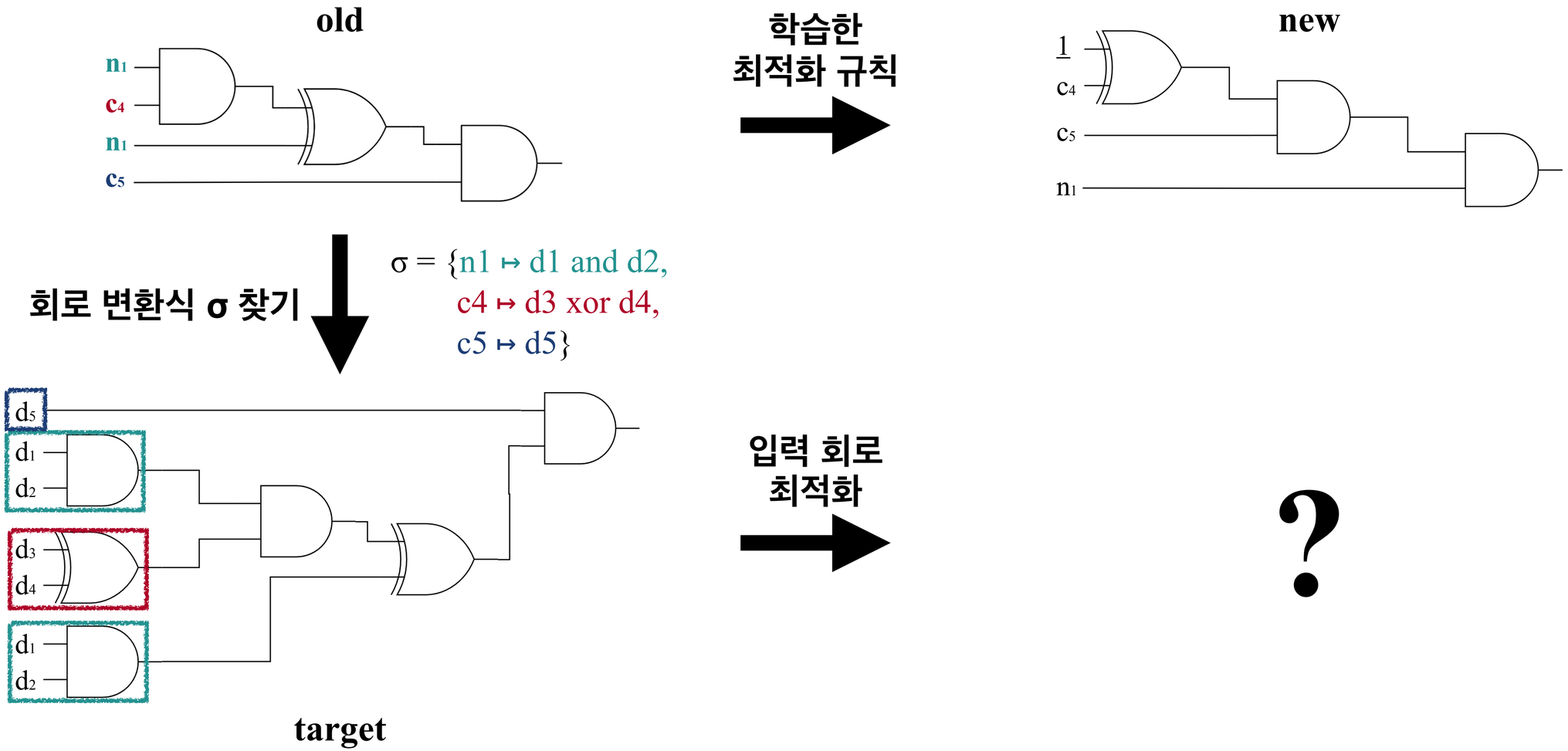
최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



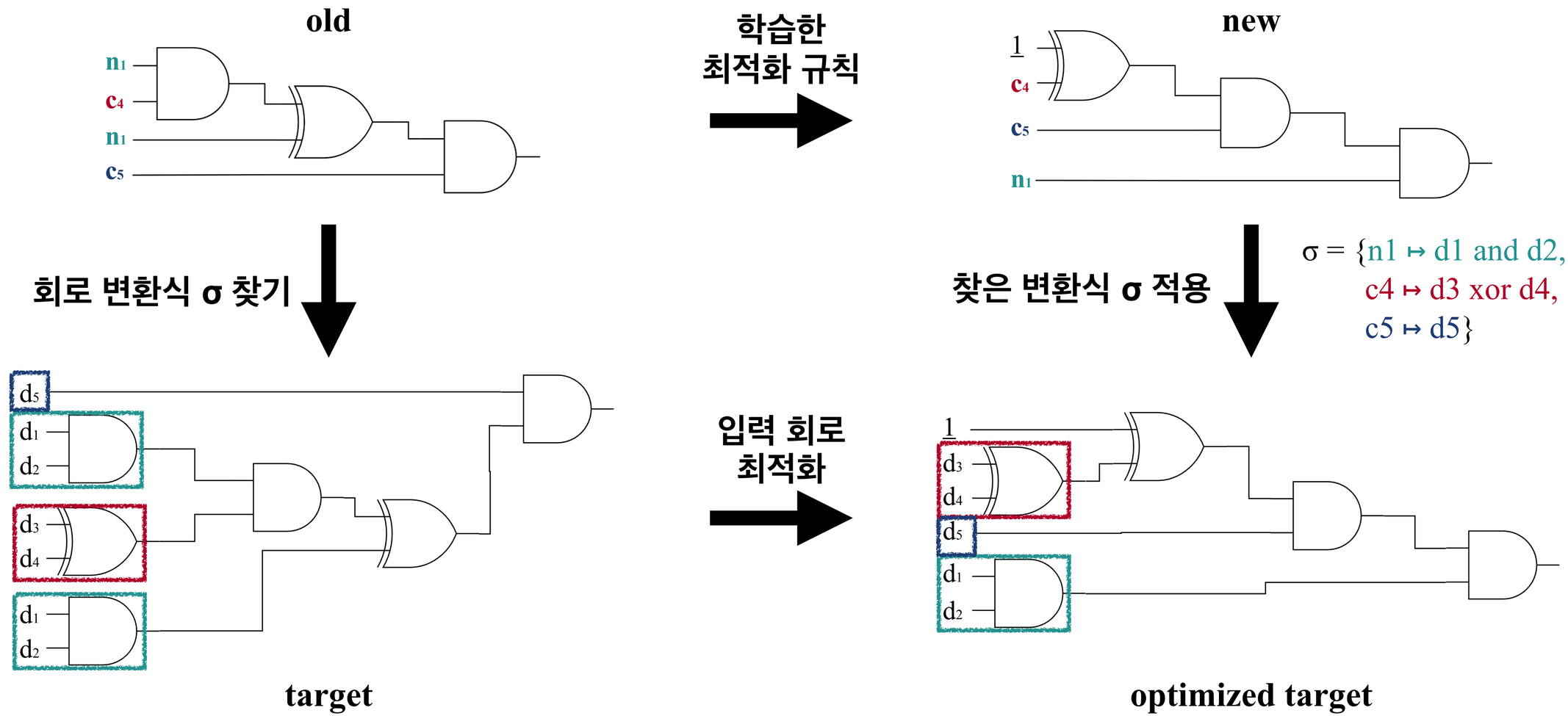
최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



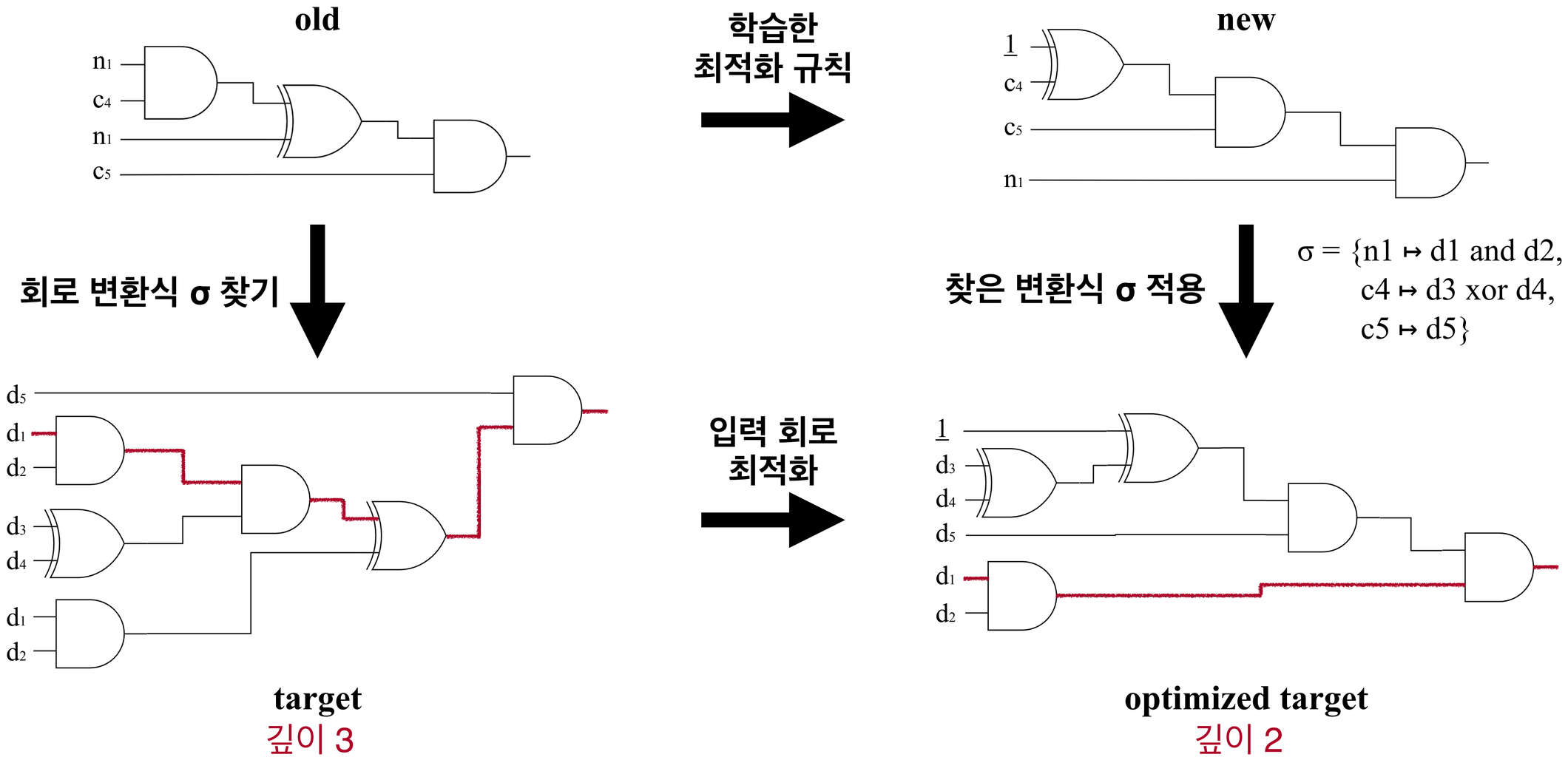
최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



최적화 규칙 유연하게 적용하기 (2/2)

회로 간소화(Normalization) + E-매칭(Equational Matching)



순서문제*Phase-Ordering Problem* : 최적화 세계의 고질적인 문제점

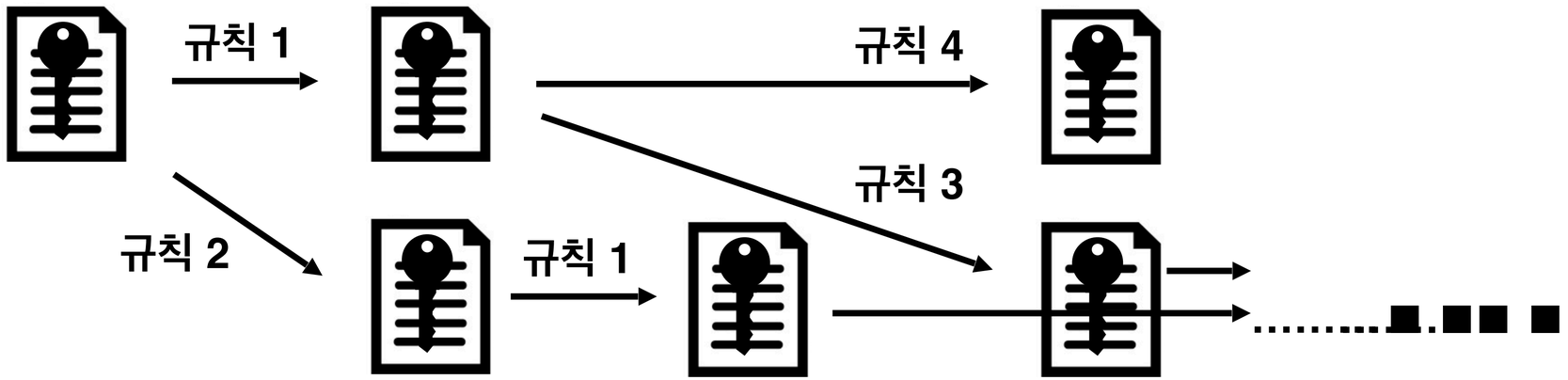
VS

순서에 따라
결과 다름

전통적인 해결책

- 정교한 휴리스틱으로 순서 정해주기(예 : LLVM 최적화순서)

- 최적화 흔적 기억하기*backtracking*(예 : n개 후보 한꺼번에 기억하면서 최적화)

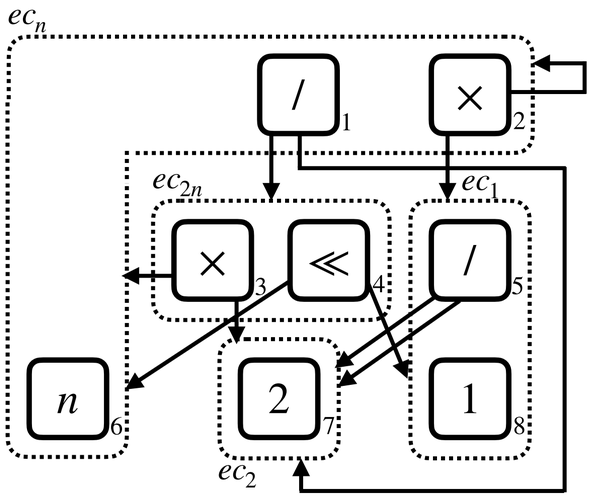


동일식 모으기 *Equality Saturation*

- 모든 경우의 수를 전부 기록하며 원본 식과 동일한 후보군 효율적으로 탐색
- 효율적인 자료구조(동일식 그래프*E-graph*)로 후보군 프로그램들을 압축적으로 표현
- 계산 비용이 비싸지만, 끝난다면 최적해 탐색 보장

동일식 모으기 도구 : 동일식 그래프*E-graph*

- 동일식 그래프 구조 = 동일식 노드*E-node* + 동일식 클래스*E-class*
  - 동일식 클래스 : 동일식 노드들의 모음
  - 동일식 노드 : 동일식 클래스들을 자식으로 가짐
- 동일식 그래프 의미
  - 동일식 노드(—) : 해당 노드에서 자식노드로 내려가며 만들 수 있는 식의 모음 (1가지가 아님)
  - 동일식 클래스(……) : 같은 클래스의 노드들은 모두 동일한 의미의 식을 생성함 (모양은 달라도)



동일식 모으기 예

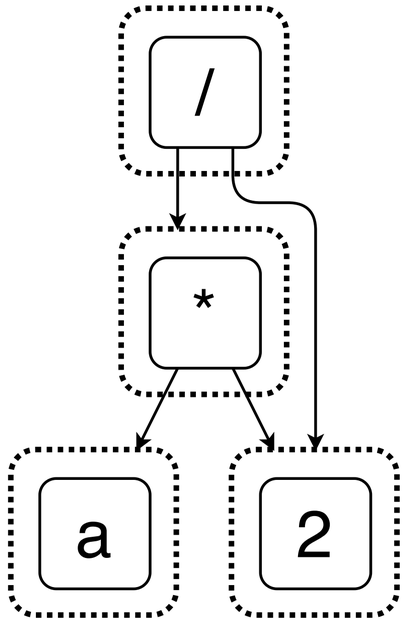
- 원본 식 $(a \times 2)/2$ 을 다음 변환 규칙들로 변환

(1) $x \times 2 \rightarrow x \ll 1$

(2) $(x \times y)/z \rightarrow x \times (y/z)$

(3) $x/x \rightarrow 1$

(4) $1 \times x \rightarrow x$



동일식 모으기 예

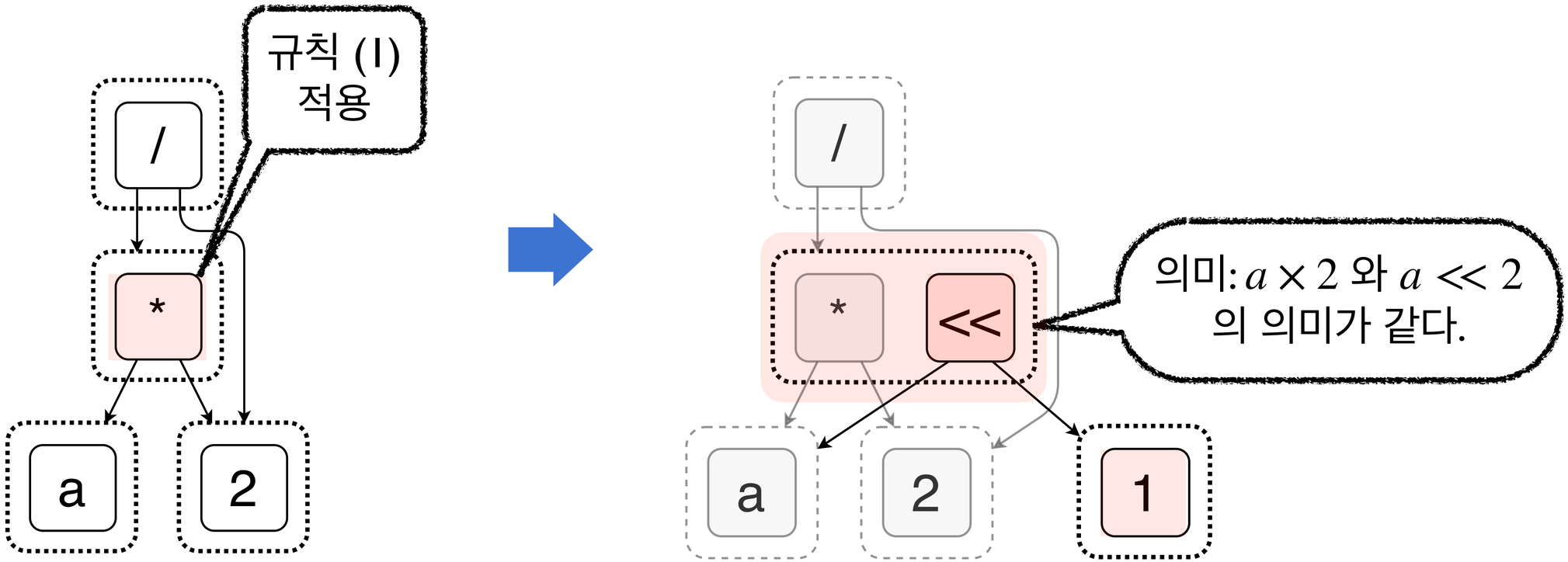
- 원본 식 $(a \times 2)/2$ 을 다음 변환 규칙들로 변환

(1) $x \times 2 \rightarrow x \ll 1$

(2) $(x \times y)/z \rightarrow x \times (y/z)$

(3) $x/x \rightarrow 1$

(4) $1 \times x \rightarrow x$



동일식 모으기 예

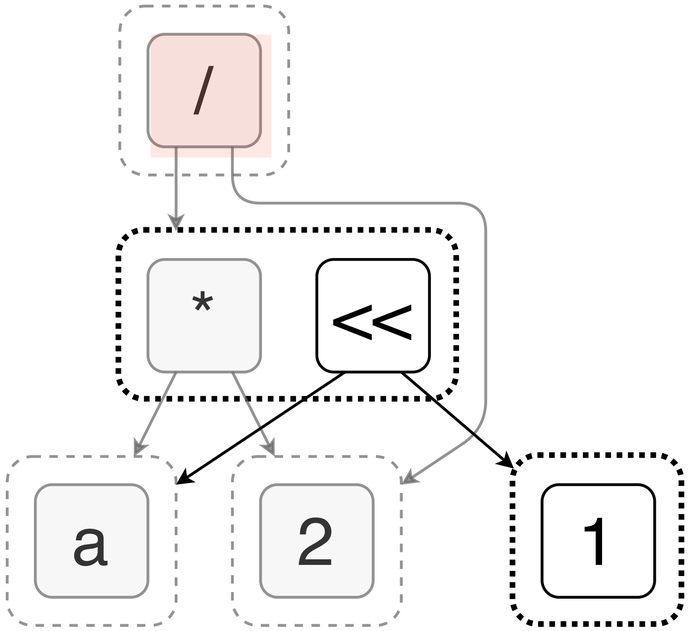
- 원본 식 $(a \times 2)/2$ 을 다음 변환 규칙들로 변환

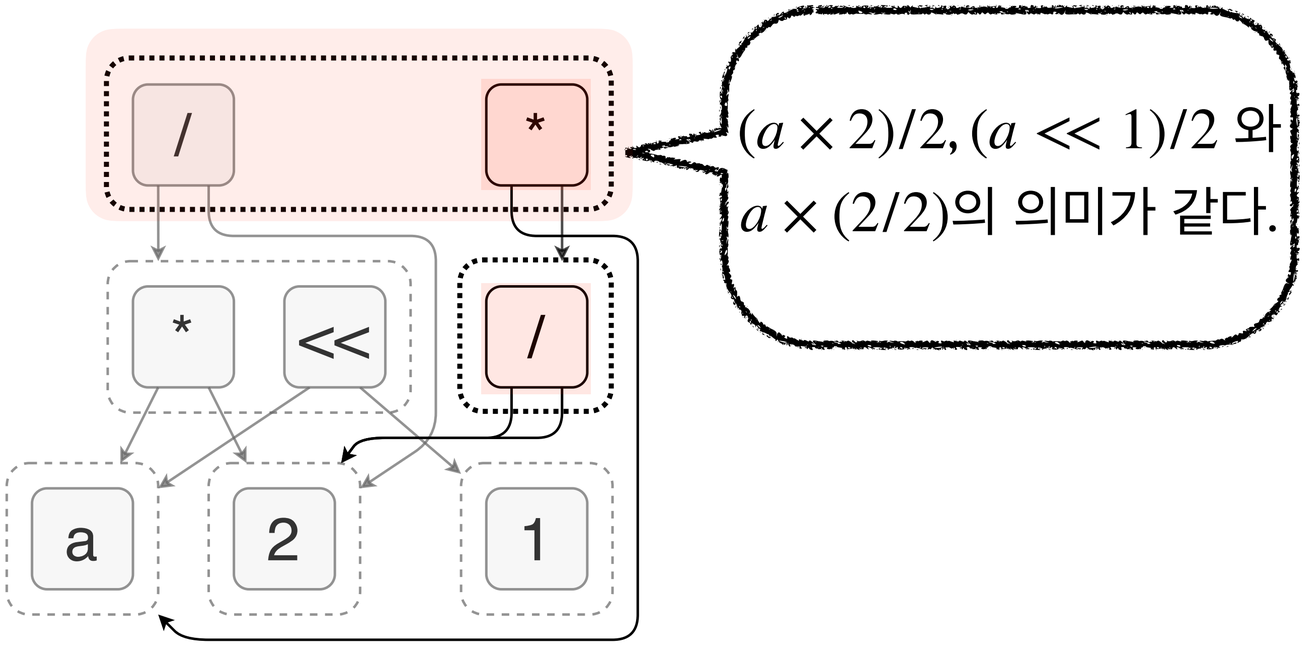
(1) $x \times 2 \rightarrow x \ll 1$

(2) $(x \times y)/z \rightarrow x \times (y/z)$

(3) $x/x \rightarrow 1$

(4) $1 \times x \rightarrow x$





동일식 모으기 예

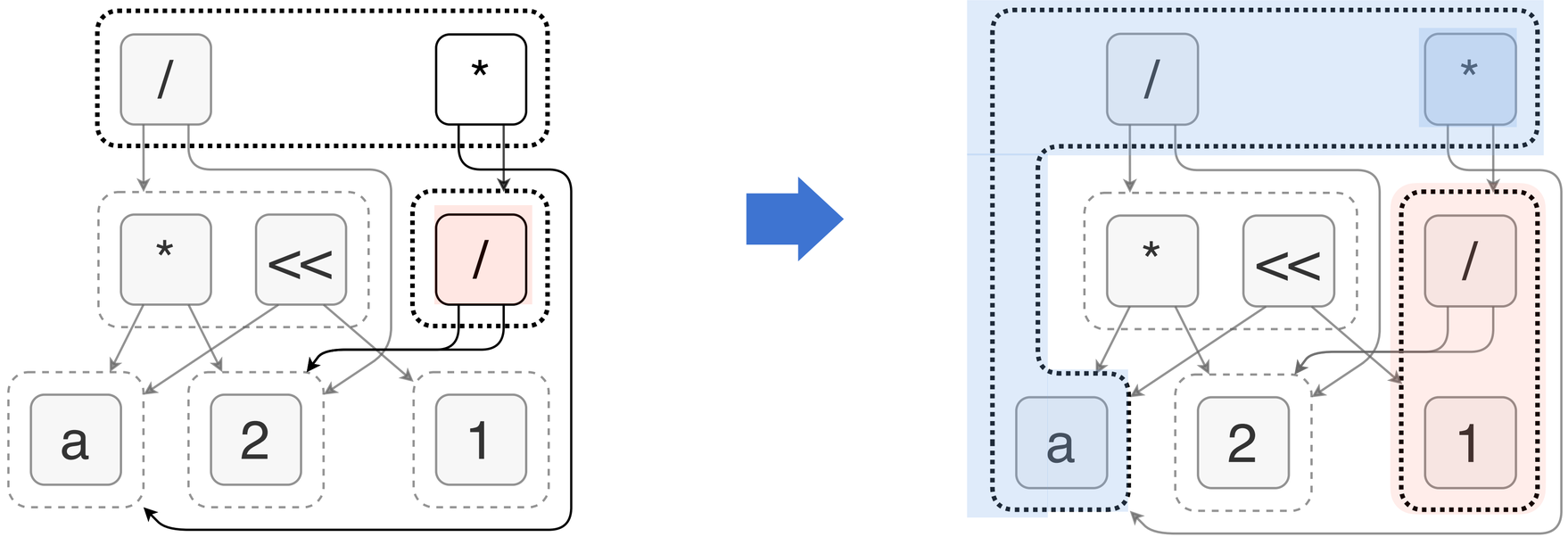
- 원본 식 $(a \times 2)/2$ 을 다음 변환 규칙들로 변환

(1) $x \times 2 \rightarrow x \ll 1$

(2) $(x \times y)/z \rightarrow x \times (y/z)$

(3) $x/x \rightarrow 1$

(4) $1 \times x \rightarrow x$



동일식 모으기 예

- 원본 식 $(a \times 2)/2$ 을 다음 변환 규칙들로 변환

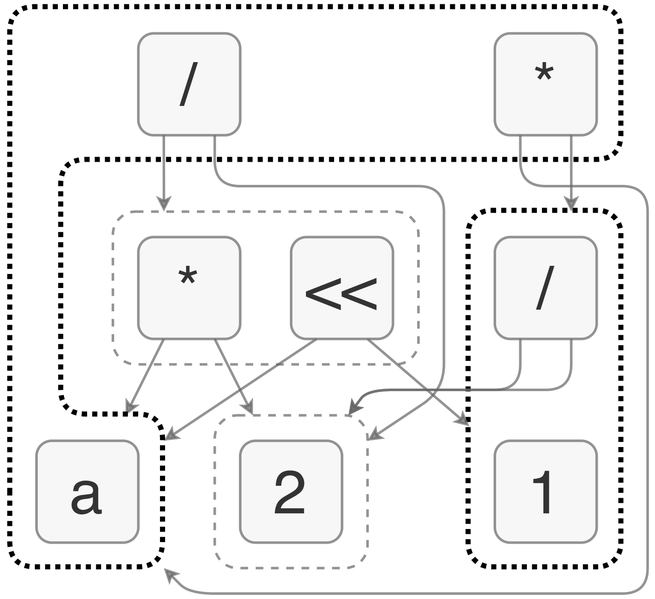
(1) $x \times 2 \rightarrow x \ll 1$ (2) $(x \times y)/z \rightarrow x \times (y/z)$

(3) $x/x \rightarrow 1$ (4) $1 \times x \rightarrow x$

- 규칙을 더 적용해도 E-graph 에 변화없음

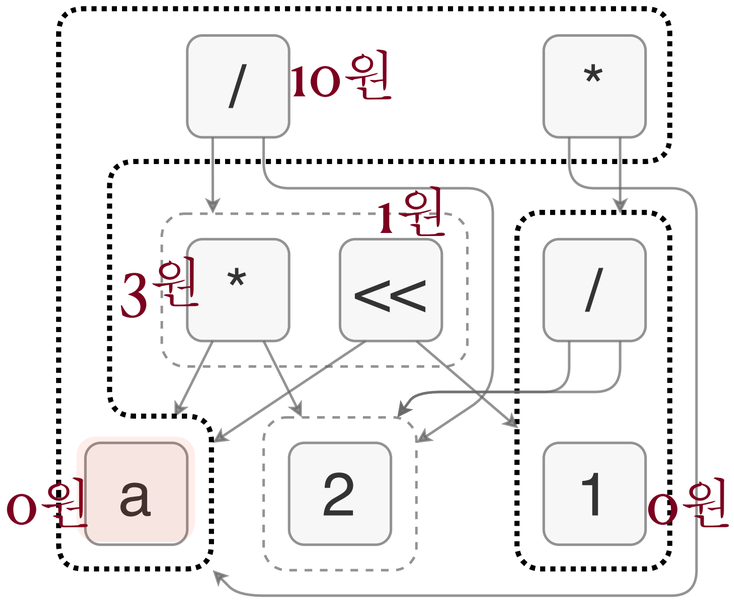
— 포화상태*Saturation*

  - 루트노드의 E-class 가 표현하는식들이 규칙들을 임의의 순서로 적용해서 얻을 수 있는 모든 식들임



모두 모은 뒤에 최적해 추출

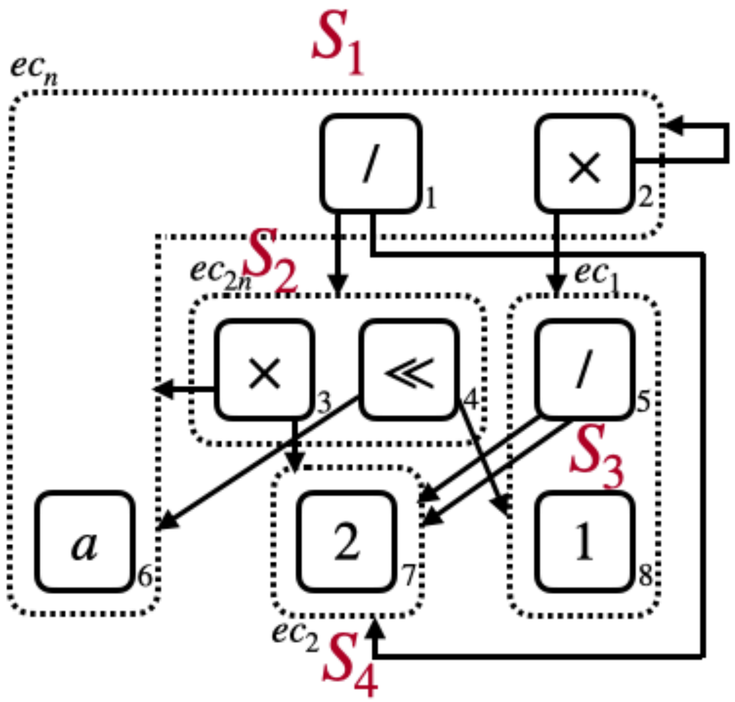
- 계속 규칙을 적용하다가 더 규칙을 적용해도 E-graph 에 변화가 없다면 *Saturation*† 가장 좋은 해를 추출
  - 동일식노드의 종류별 계산비용을 지정해주면 자동으로 추출가능
  - 더 복잡한 계산비용 사용시 선형계획법*Integer Linear Programming*을 쓰기도



† 동일식 모으기의 종료는 항상 보장되지는 않음.

동일식 모으기의 근본적 의미

- E-graph ≅ 의미가 동일한 식들을 표현하는 문법
  - E-class ≅ 비말단기호*non-terminal*, E-node ≅ 문법규칙*production rule*
- 즉, 의미가 동일한 식들을 표현하는 문법을 유추하는 것*Grammar induction*



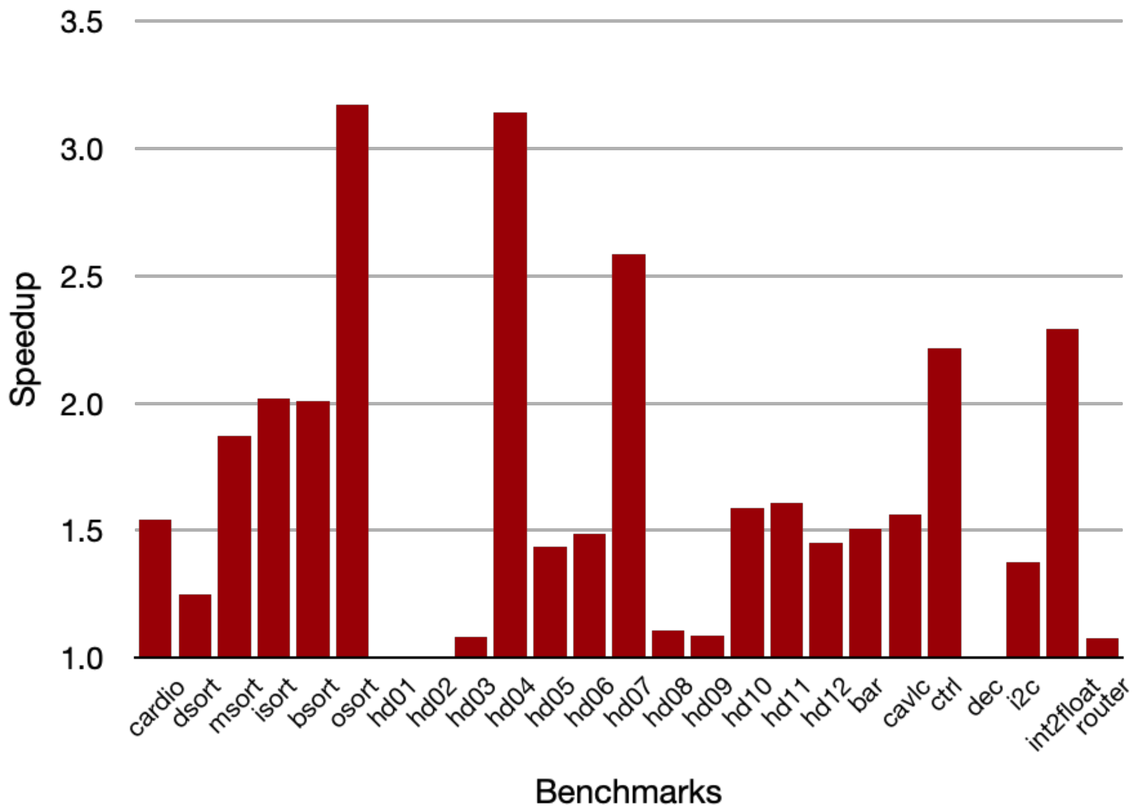
$$
\begin{aligned}
S_1 &\rightarrow S_2 \;/\; S_4 \\
&\mid S_1 * S_3 \\
&\mid a \\
S_2 &\rightarrow S_1 \;<<\; S_3 \\
&\mid S_1 * S_4 \\
S_3 &\rightarrow S_4 \;/\; S_4 \\
&\mid 1 \\
S_4 &\rightarrow 2
\end{aligned}
$$

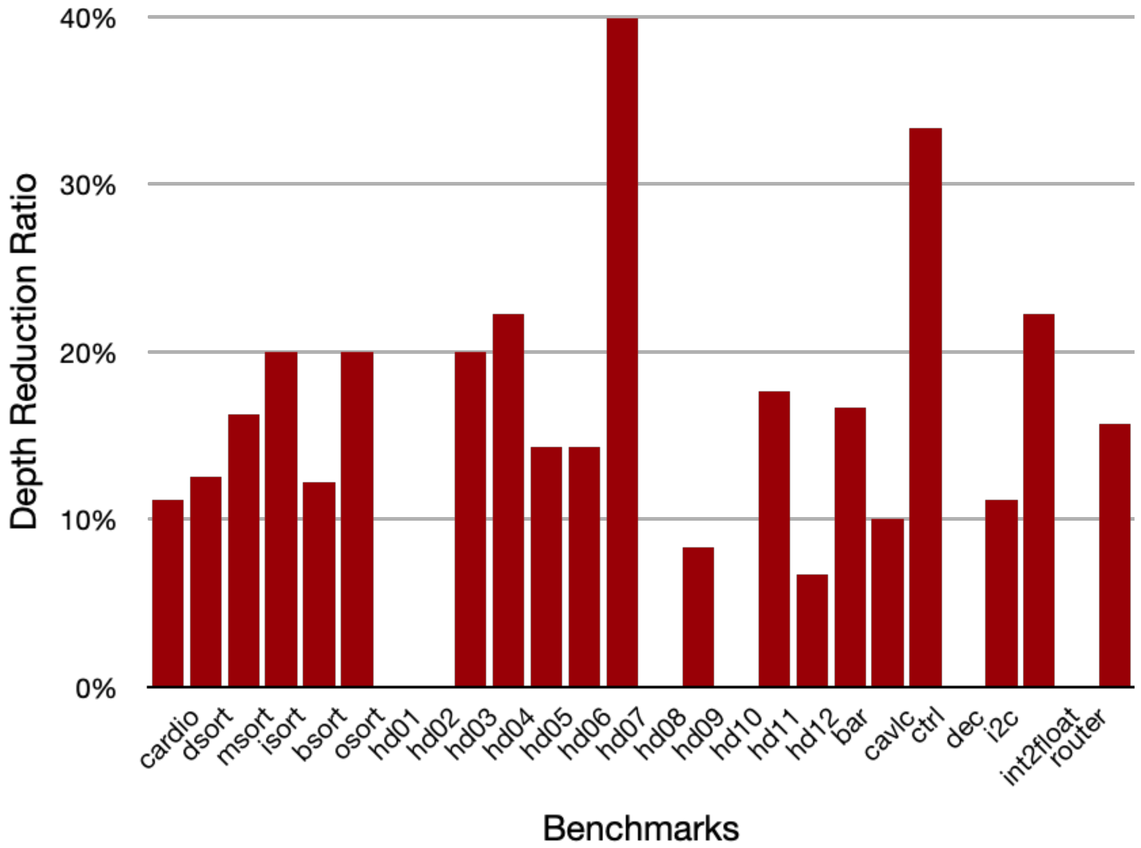
실험

- 작은 회로는 동일식 모으기, 큰 회로는 식 다시쓰기
- 4가지 카테고리, 25개의 동형암호 알고리즘
  - Cingulata benchmarks
  - Sorting benchmarks
  - Hackers Delight benchmarks
  - EPFL benchmarks
- 비교대상: Cingulata
  - 전문가가 작성한 최적화 규칙들을 사용하는 동형암호 컴파일러

실험결과

- 수동 작성된 최적화 규칙을 사용하는 동형암호 컴파일러 Cingulata 대비 **평균 2배, 최대 3.1배 빠른 성능** 달성 (곱셈깊이 최대 40% 감소)

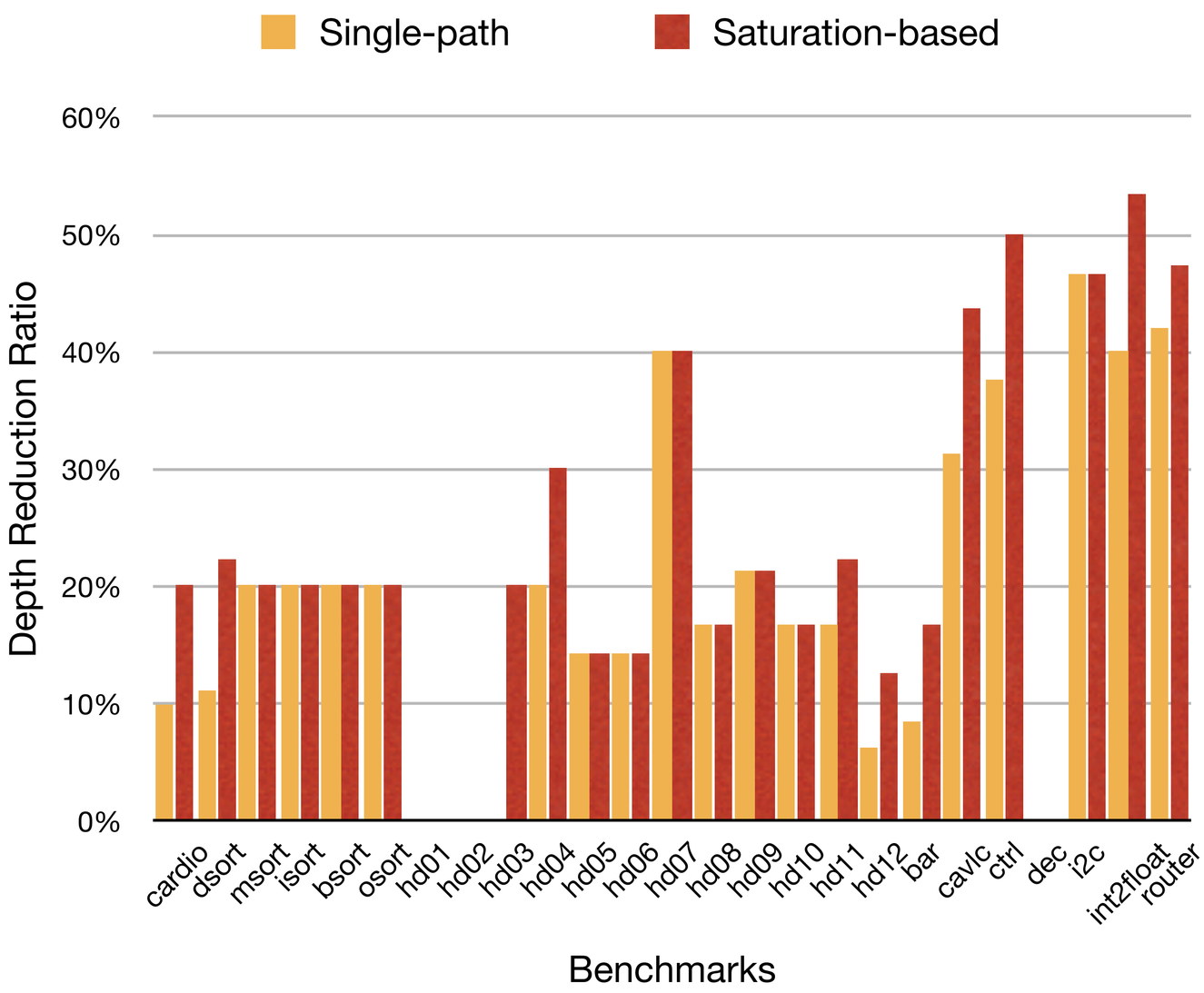




동일식 모으기의 효과

최적화 성공률 증가 : 19 → 22개 최적화 성공

- 최적화 성능 증가 (실행시간 감소율) : x2.03 → x2.26
- 최적화 성능 증가 (곱셈깊이 감소율) : 21.9% → 25.1%



차례

- 최적화 사례 1 : 프로그램 합성 기반 동형암호 최적화

- **최적화 사례 2 : 프로그램 합성 기반 프로그램 역난독화**
  - Jaehyung Lee and Woosuk Lee, *Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting*, **ACM CCS 2023**
  - Jaehyung Lee, Seoksu Lee, Eunsun Cho and Woosuk Lee, *Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Equality Saturation*, **IEEE TDSC (Submitted)**
- 두 사례로 부터 관찰한 것들

- 기반 원천 기술 : 고성능 프로그램 합성 기술

Mixed Boolean Arithmetic (MBA)

- 변수들 간 논리연산 (AND, OR, XOR 등) 과 산술연산 (사칙연산, 쉬프트 등) 이 얽혀있는 연산식

- 예 : $8458(x \vee y \wedge z)^3 \left((xy) \wedge x \vee t\right) + x + 9(x \vee y)yz^3$

- MBA 난독화 : 프로그램에 등장하는 임의의 연산식을 복잡한 MBA 식으로 변환하는 과정

MBA 난독화의 인기

- 난독화 및 난독화된 코드의 실행을 위한 비용이 쌈
  - 기본적인 연산들만 추가되고 실행흐름이 바뀌지 않음 (유저/시스템 함수 추가 호출 등 없음)
- 이론적 탄탄함: 임의의 비트연산식을 복잡한 MBA 식으로 변환하는 방법이 무한히 많음. 원본 식을 유추하는 것은 NP-hard.
- 여러 상용 도구에 사용됨
  - 코드 난독화: Tigress, VMProtect 등, DRM
  - 악성웨어들에도 사용되는 중

기존 접근법들

- **식 다시쓰기** : SSPAM [Eyrolles et al. 2016]

다양한 MBA 난독화 규칙 다루지 못함

- **프로그램 합성** : Syntia [Blazytko et al. 2017], QSynth [David et al. 2020], Xyntia [Menguy et al. 2021]

결과의 올바름 보장 X

- **신경망 추론** : NeuReduce [Feng et al. 2020]

크고 복잡한 MBA 식 다루지 못함

- **대수적 방법** : MBA-Solver [Xu et al. 2021], SiMBA [Reichenwallner et al. 2022]

특정 유형의 MBA식만 다룰 수 있음

기존 접근법들

- 식 다시쓰기 : SSPAM [Eyrolles et al. 2016]

다양한 MBA 난독화 규칙 다룰지 못함

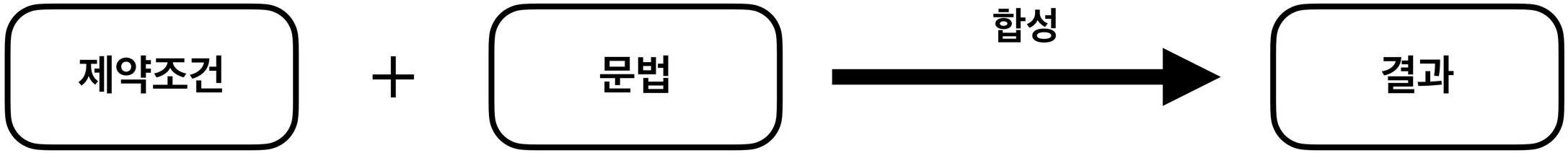
우리의 목표:

- **안전성** : 역난독화 결과의 올바름 보장
- **일반성** : 임의의 MBA 식 다룰 수 있어야.
- **유연성** : 임의의 난독화 규칙들이 사용되어도 문제없이 역난독화
- **확장성** : 거대한 MBA 식도 단순화 할 수 있어야

- 대수적 방법 : MBA-Solver [Xu et al. 2021], SiMBA [Reichenwallner et al. 2022]

특정 유형의 MBA식만 다룰 수 있음

프로그램 합성 기반 MBA 역난독화




```
(((((~ e) + 1) & (~ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a))))) + (((~ e) + 1) & (~ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a)))))) - (((~ e) + 1) ^ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a))))) …
```

$$
\begin{aligned}
S &\rightarrow\ !S \mid S \oplus S \mid S \wedge S \\
&\mid\ S \vee S \mid -S \mid S + S \\
&\mid\ S \times S \mid S - S \mid S >> S \\
&\mid\ S << S \mid V \mid C \\
V &\rightarrow\ \texttt{b} \mid \texttt{e} \mid \cdots \\
C &\rightarrow\ \texttt{0x00} \mid \texttt{0x01} \mid \cdots
\end{aligned}
$$

1 + a

원본과 똑같은 의미,

가능한 작은 것

임의의 MBA 식 표현 가능

한계점 : 합성기 성능(Scalability)

(((((~ e) + 1) & (~ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a))))) + (((~ e) + 1) & (~ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a)))))) - (((~ e) + 1) ^ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a))))) …

합성

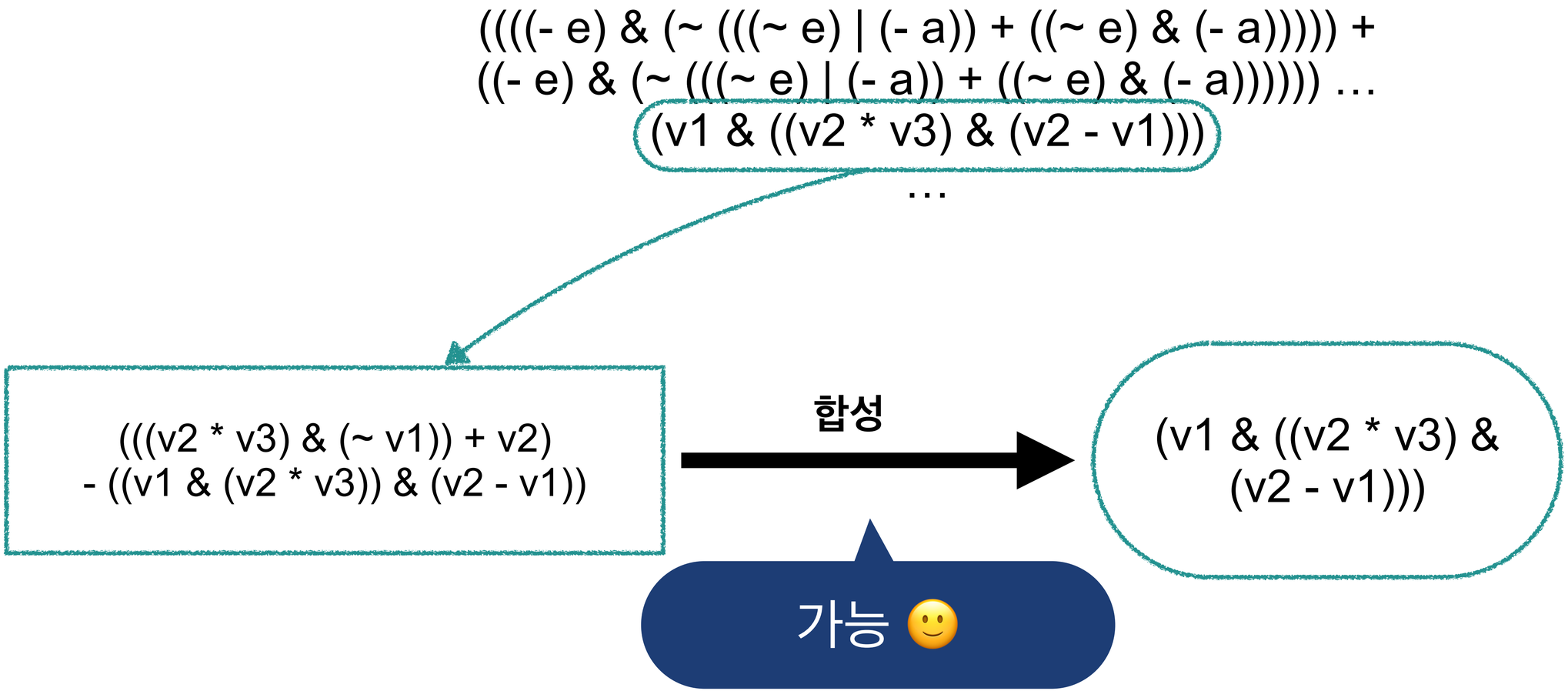
1 + a

너무 느림

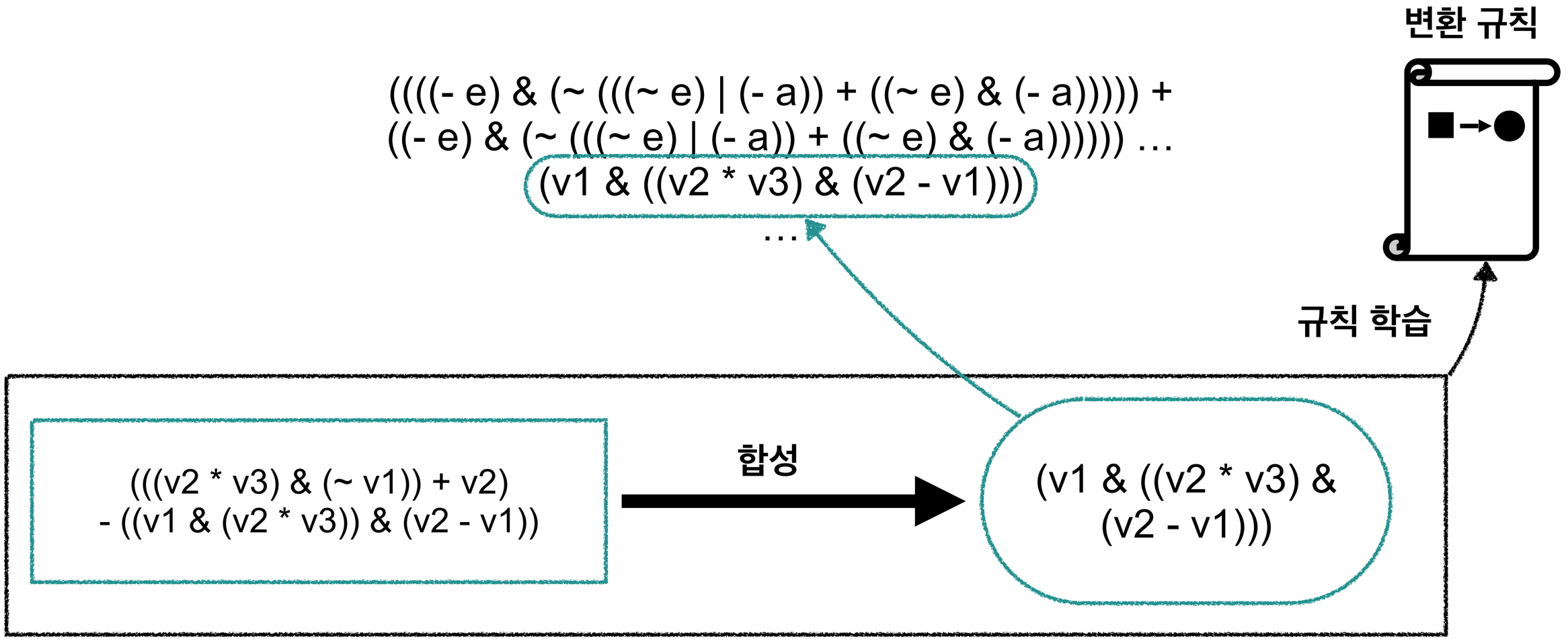
난독화 식 크기 ⬆ ⇨ 합성 성능 ⬇

해결책 1 : 부분적으로 합성하기

- 일부분 선택



해결책 2 : 성공적인 최적화 패턴 기억하고 적용



수집된 역난독화 패턴 예

$$((((((b - e) - ((b \mid (\sim e)) << 1)) - 2) \mid b) - ((((b - e) - ((b \mid (\sim e)) << 1)) - 2) \,\&\, b)) \mid ((((\sim e) \mid e) + e) + 1)) \;\rightarrow\; e$$

$$(((d \,\&\, d) * (d \mid d)) + ((d \,\&\, (\sim d)) * ((\sim d) \,\&\, d))) \;\rightarrow\; ((d * d) \,\&\, (d * d))$$

$$(((b - e) - ((b \mid (\sim e)) + (b \mid (\sim e)))) - 2) \;\rightarrow\; ((b \,\hat{}\, e) \,\&\, (b \mid e))$$

$$((- (b \mid 1)) \,\&\, (\sim b)) \;\rightarrow\; (\sim b)$$

동형암호 최적화와 공통점 / 차이점

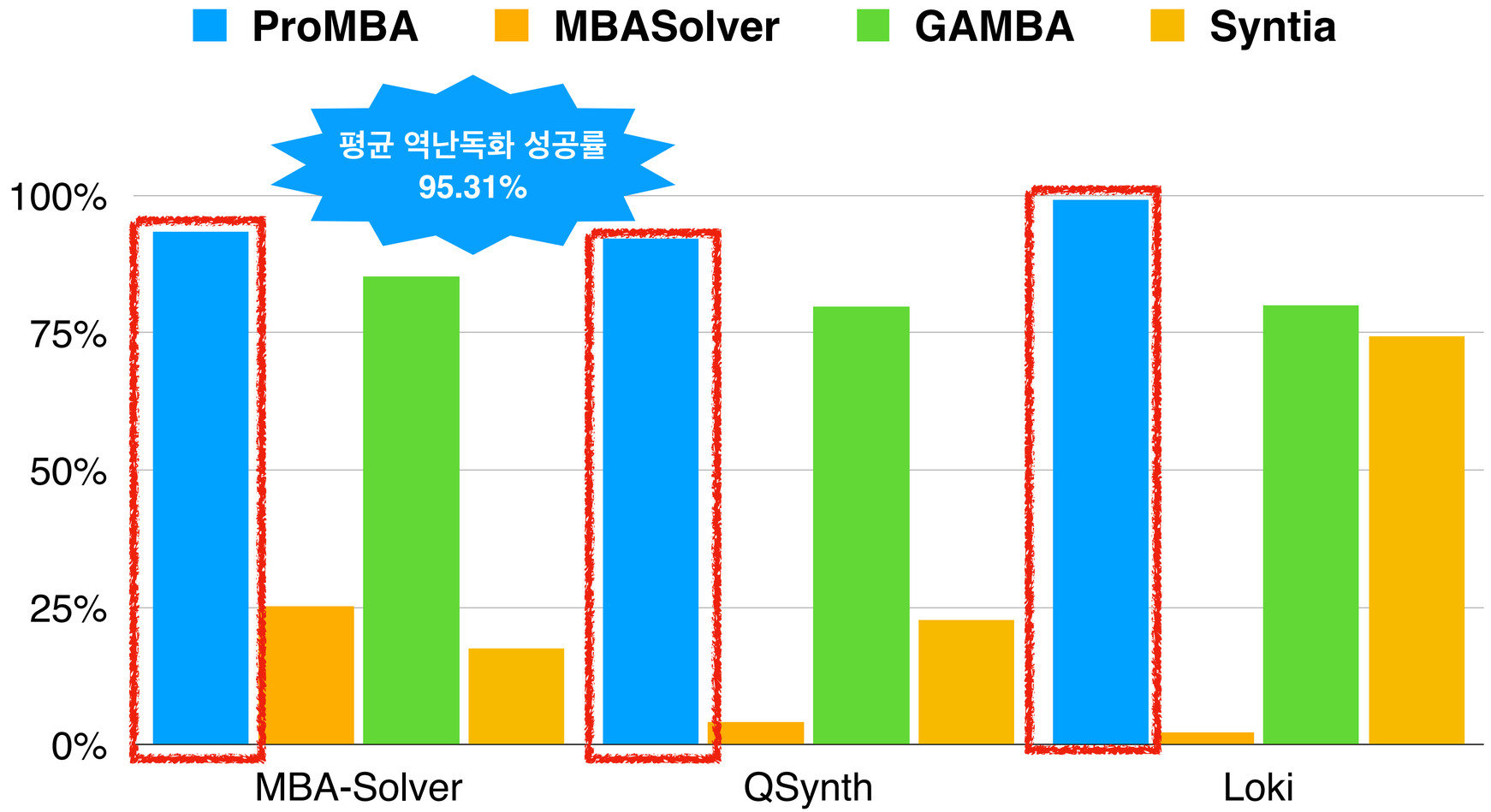
- **공통점** : 부분적으로 합성 → 규칙 학습 → 식 다시쓰기 + 동일식 모으기
- **주된 차이점** : 오프라인 규칙 학습 없이 온라인으로 바로 규칙 학습 및 적용
  - MBA 난독화에 사용되는 규칙이 너무나 다양 — 오프라인으로 배운 규칙이 새로운 MBA식 역난독화에 쓸모 없음
  - 합성기의 발전으로 규칙 합성이 빠르게 됨
- 그 외 차이점: 특정 유형의 MBA식(linear MBA)은 대수적 방법으로 변환. 교체대상 부분식 선택하는 방법 등

실험

- 툴 ProMBA 구현. 식 다시쓰기 끝까지. 이 후 동일식 모으기 적용
- 기존 연구에서 사용된 4000여개의 MBA 난독화 된 식들 대상
  - 3개 카테고리. 작은~큰 그기 다양한 원본 식들을 난독화
- 비교대상
  - MBASolver [PLDI ‘22] : 올바름 보장. 대수적 방법
  - Syntia [USENIX ’17] : 올바름 보장 X. 휴리스틱 기반
  - GAMBA [WORMA’23] : 올바름 보장. 대수적 방법 + 휴릭스틱
- 성공여부: 역난독화 결과와 원본 의미 동일 + 결과 크기 ≤ 원본 크기

결과

- 평균 역난독화 성공률 95.3%로 타 도구 압도 (13%, 82.5%, 39.4%)



동일식 모으기의 효과

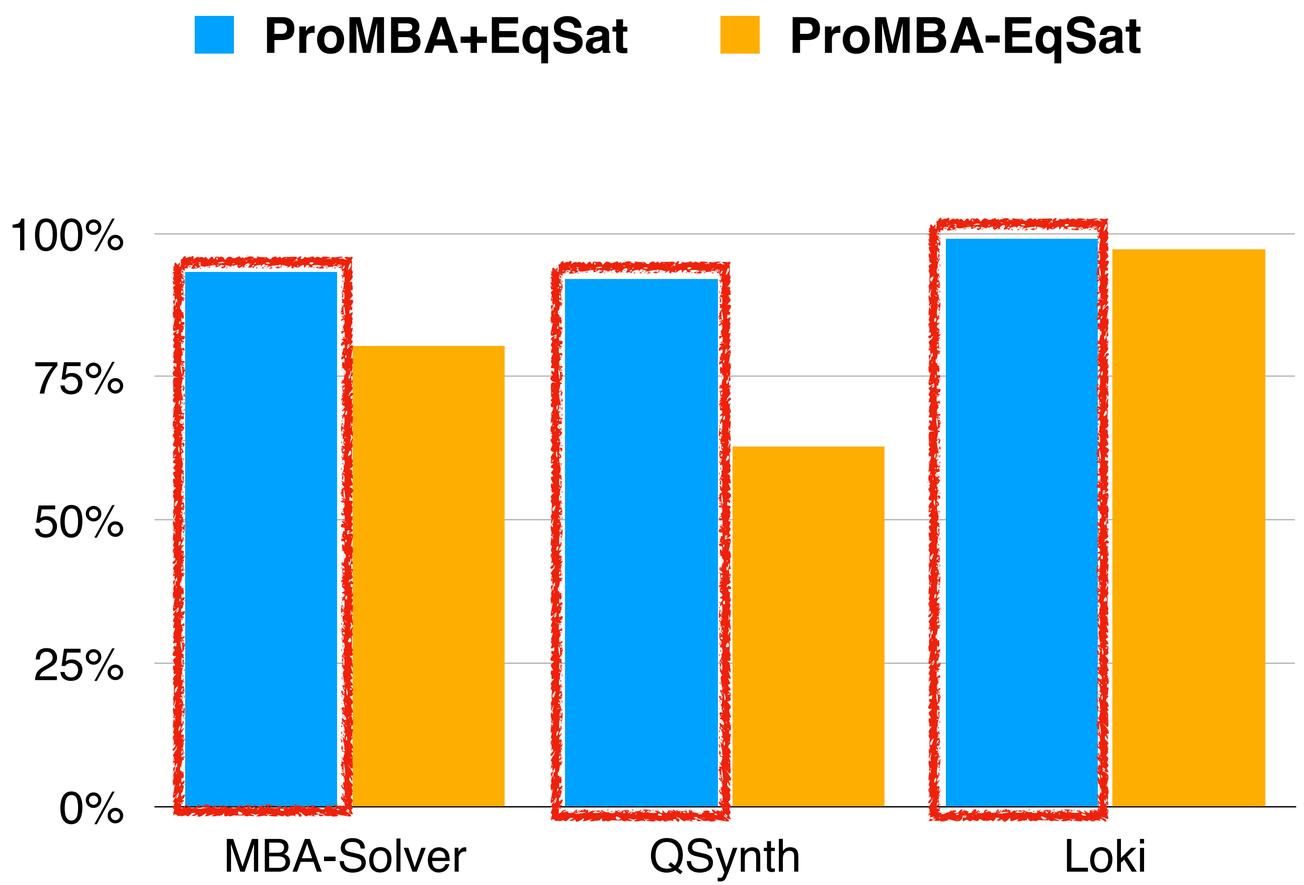
- 역난독화 성공률 증가 :

  84% → 95%

- 역난독화 결과의 평균

  크기 감소 : 9.4 → 7.9

  (AST노드 갯수 기준)



차례

- 최적화 사례 1 : 프로그램 합성 기반 동형암호 최적화
- 최적화 사례 2 : 프로그램 합성 기반 프로그램 역난독화
- **두 사례로 부터 관찰한 것들**
- 기반 원천 기술 : 고성능 프로그램 합성 기술

관찰 1 : 식 다시쓰기, 동일식 모으기 둘다필요 (1/2)

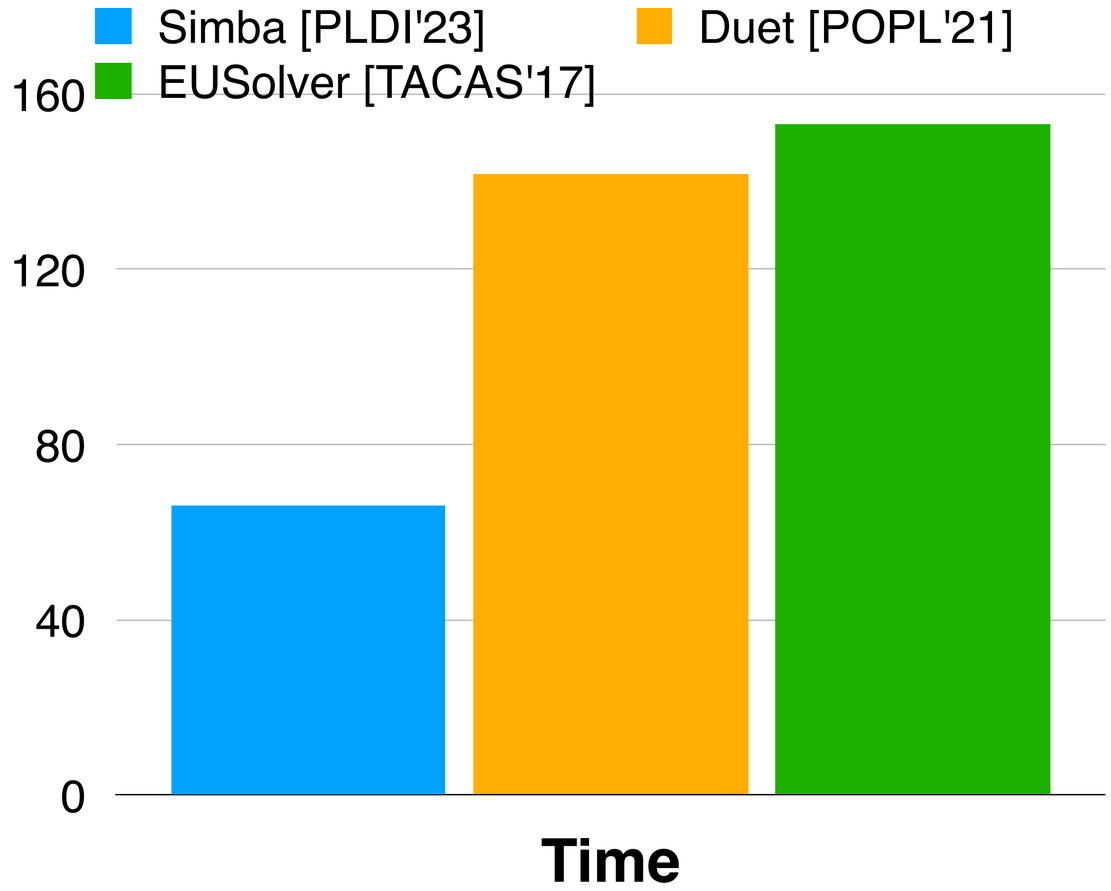
- 동일식 모으기는 식 다시쓰기의 한계*Phase-ordering problem*를 극복한다고 알려짐. 그러나 실제로는 동일식 모으기 만으로는 부족
- 동일식 모으기의 가장 큰 문제는 비용
- 동형암호 최적화의 경우 — 식 다시쓰기 없이 처음부터 동일식 모으기만 적용 시, 큰 동형암호 회로(깊이 25이상)들은 메모리(256G) 부족으로 실패. 작은 회로도 saturation 안되기도 (12시간 시간제한)
- MBA 역난독화의 경우 — 식 다시쓰기 없이 처음부터 동일식 모으기만 적용 시 (단, 높은 비용을 피하기 위해 saturation 전에 적절히 끝냄), 식 다시쓰기 이 후 동일식 모으기 적용 대비 낮은 성공률 (92% → 61.8% 로 감소)

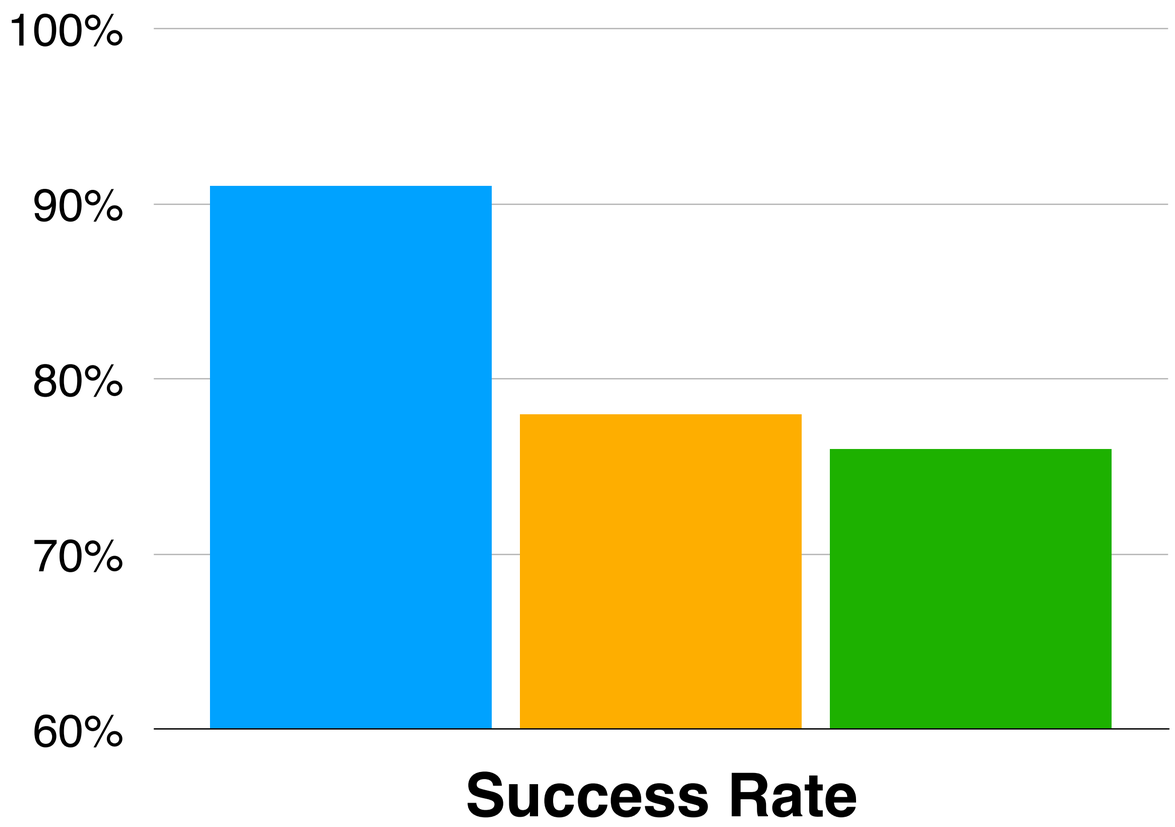
관찰 1 : 식 다시쓰기, 동일식 모으기 둘다필요 (2/2)

- 동일식 모으기 수행 시 원본식이 클수록, 규칙이 많을수록 탐색범위 및 계산비용 증가
- 그러므로 식 다시쓰기를 끝까지 한 후, 동일식 모으기 적용이 유리
  - 원본식을 작게 만들어 탐색범위 감소. 탐색에 방향성 부여

관찰 2 : 프로그램 합성의 성능이 중요 (1/2)

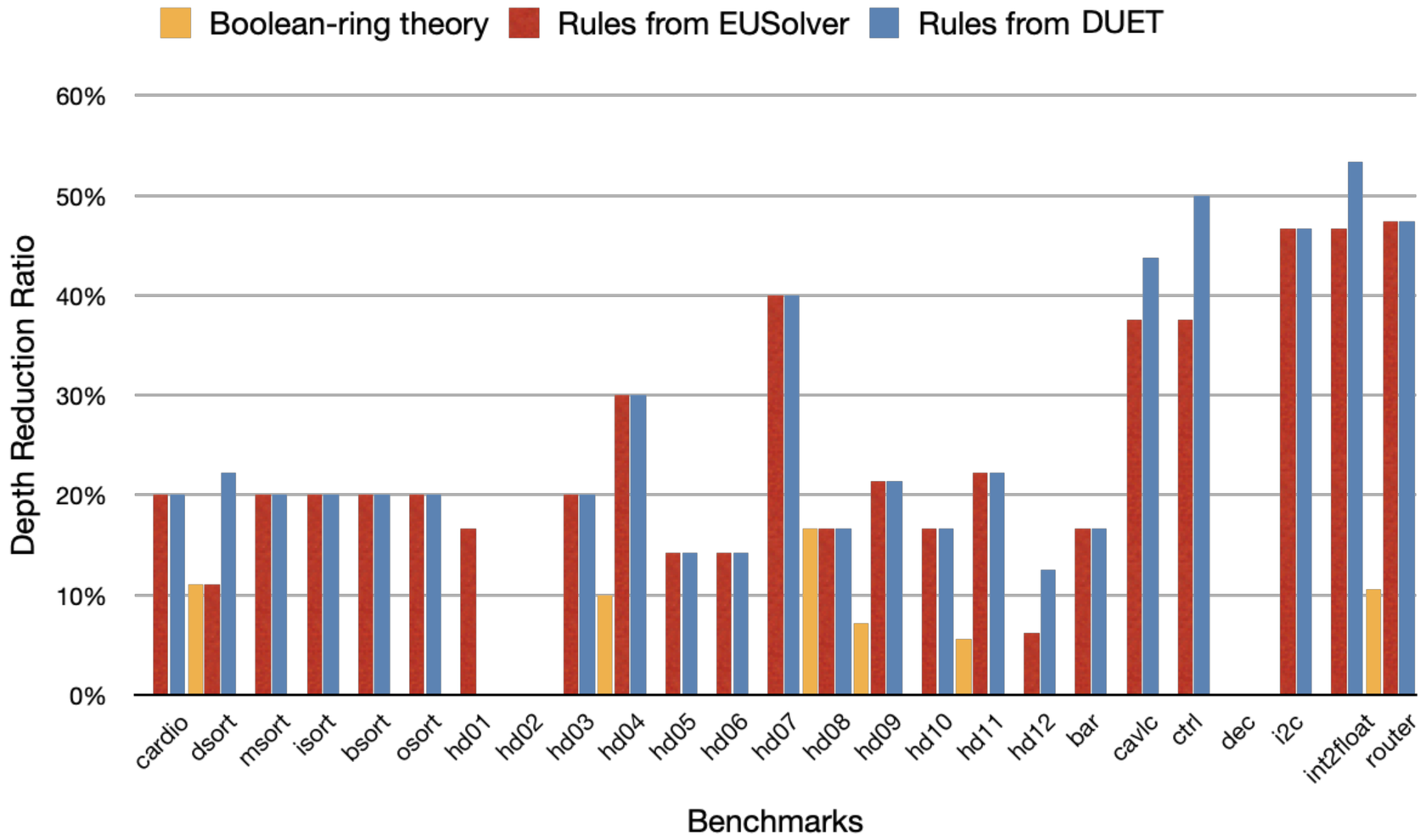
- 더 좋은 합성기로 찾은 규칙이 더 좋은 최적화
- MBA 역난독화의 경우 (합성기 성능 Simba > Duet > EUSolver)





관찰 2 : 프로그램 합성의 성능이 중요 (2/2)

- 동형암호 최적화의 경우 (합성기 성능 Duet > EUSolver)

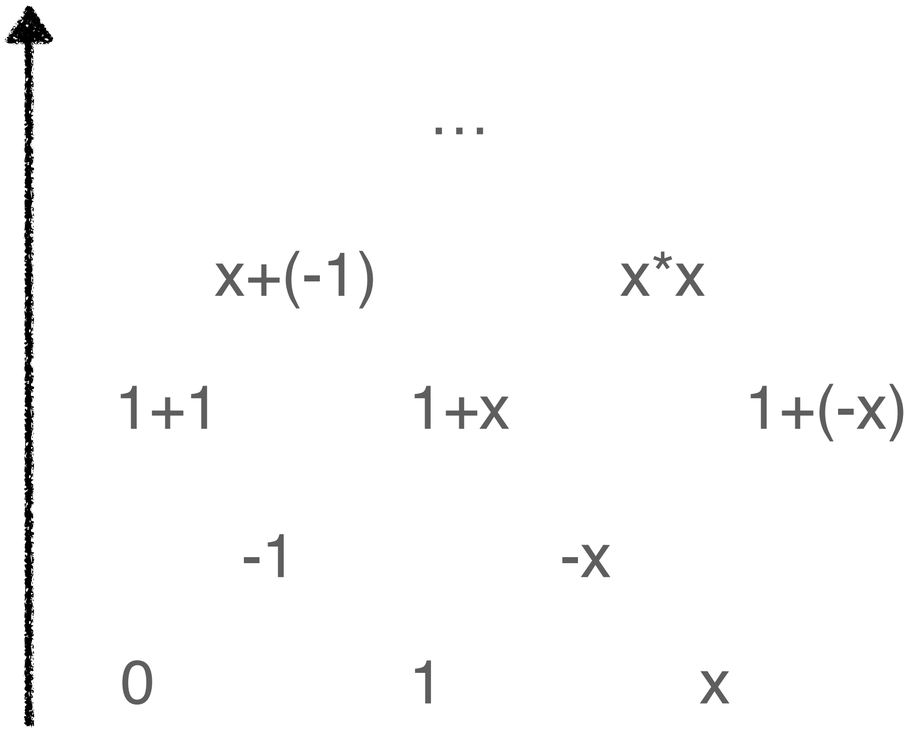


차례

- 최적화 사례 1 : 프로그램 합성 기반 동형암호 최적화
- 최적화 사례 2 : 프로그램 합성 기반 프로그램 역난독화
- 두 사례로 부터 관찰한 것들
- **기반 원천 기술 : 고성능 프로그램 합성 기술**
  - Yongho Yoon, Woosuk Lee, and Kwangkeun Yi, Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. **ACM PLDI 2023**

합성의 두 갈래 — 상향식 (Bottom-Up)

장점: 실행가능한 완성된 후보들이 탐색되므로 동적분석 기반 최적화 가능
(실행결과 같은 중복후보들 제거)
단점: 전체 프로그램 얼개에 대한 정보없이 목표와 상관없는 많은 후보들 탐색



$$
\begin{array}{rcl}
E & \rightarrow & 0 \\
 & | & 1 \\
 & | & x \\
 & | & E + E \\
 & | & E * E \\
 & | & -E
\end{array}
$$

합성의 두 갈래 — 하향식 (Top-Down)

0 1 x E+E E*E -E

1+1 1+x x+x E*(-E)

-E+E E*E+E E*(E+E)

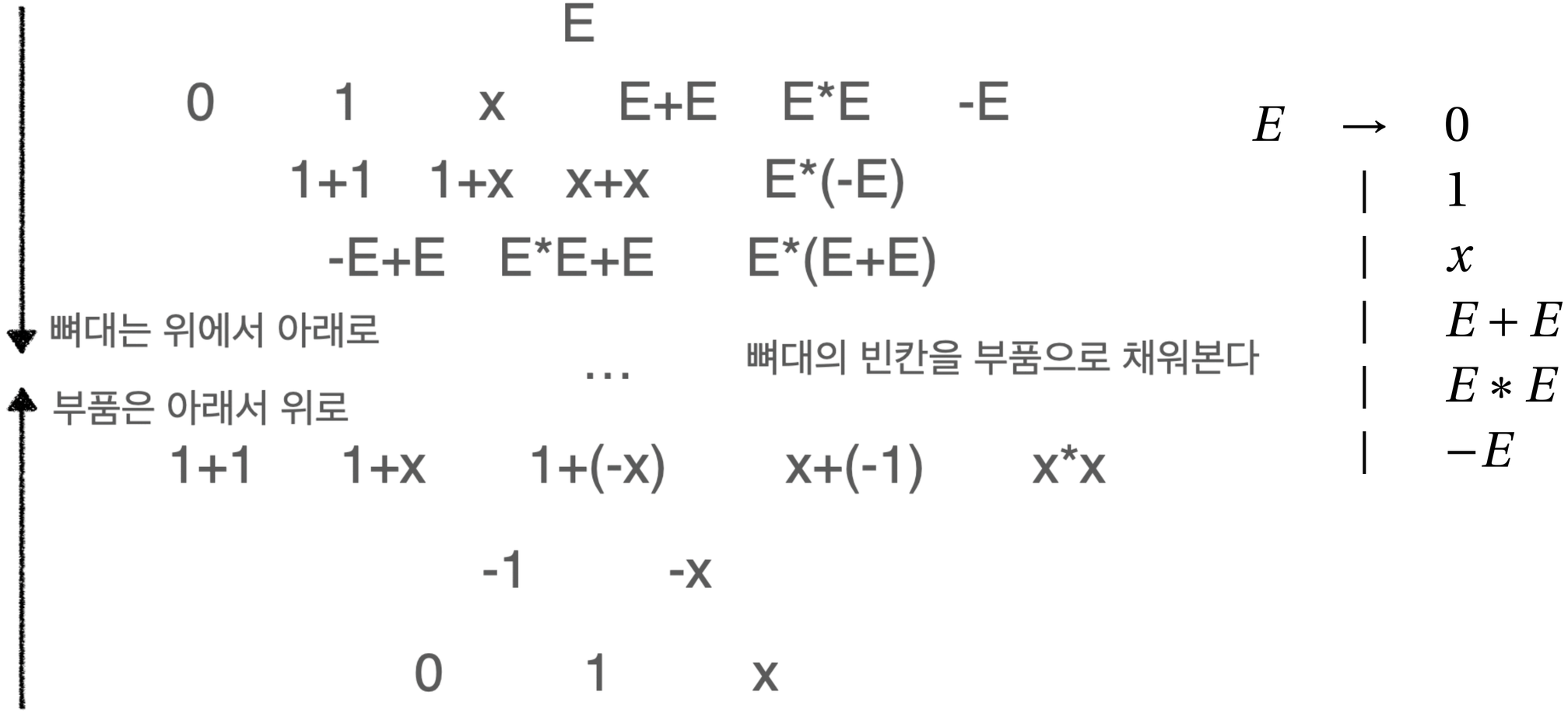
…

$$
\begin{array}{rcl}
E & \rightarrow & 0 \\
 & | & 1 \\
 & | & x \\
 & | & E + E \\
 & | & E * E \\
 & | & -E
\end{array}
$$

장점: 전체 프로그램 얼개가 탐색되므로 정적분석을 활용한 최적화 (불가능한 후보쳐내기)

단점: 실행할 수 없는 후보들만 탐색되므로 동적분석 기반 최적화 불가

양방향 탐색



$$
\begin{array}{rcl}
E & \rightarrow & 0 \\
 & | & 1 \\
 & | & x \\
 & | & E + E \\
 & | & E * E \\
 & | & -E
\end{array}
$$

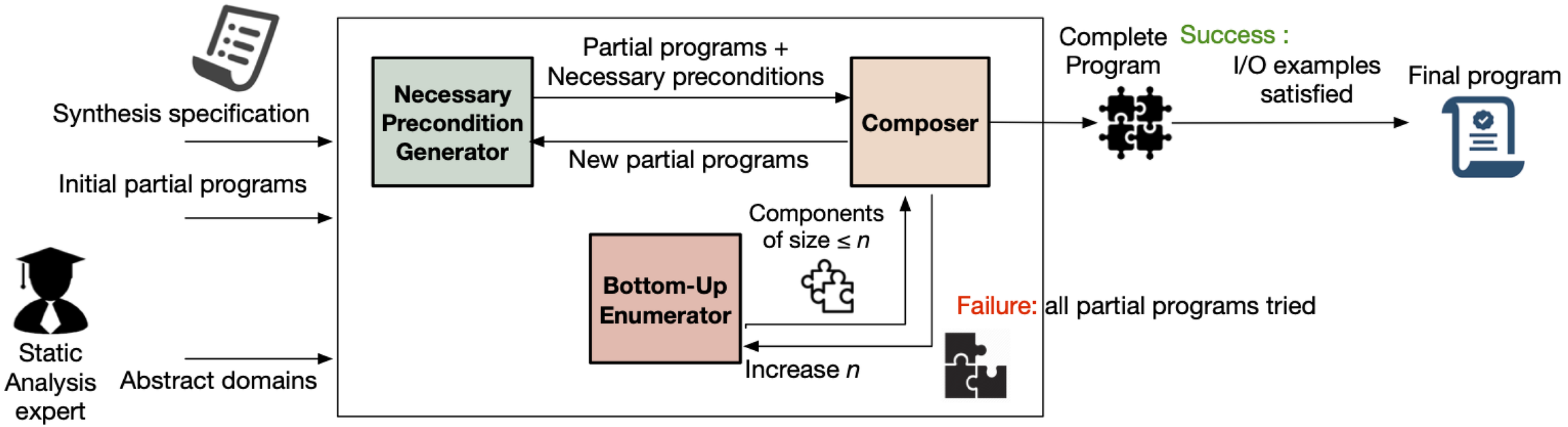
*(POPL'21) Woosuk Lee, "Combining the Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis"

양방향 탐색 + 정적분석

- 싹수없는 뼈대를 빨리 판단해서 버리면 유리함
  - 싹수없는 = 아직 미완성 프로그램이지만 빈 칸에 어떤 부품을 끼워봐도 조건을 만족하지 못할 것이 확실한
  - 후보 부품 갯수가 많을 수록 버리기의 효과 큼

- “**매우 정교한**” 정적분석(이하, **싹수분석**)으로 잘 할 수 있지 않을까?
  - 입력: 조건(입출력 쌍)과 미완성 프로그램
  - 출력: “정답일수도 있음” or “싹수없음”

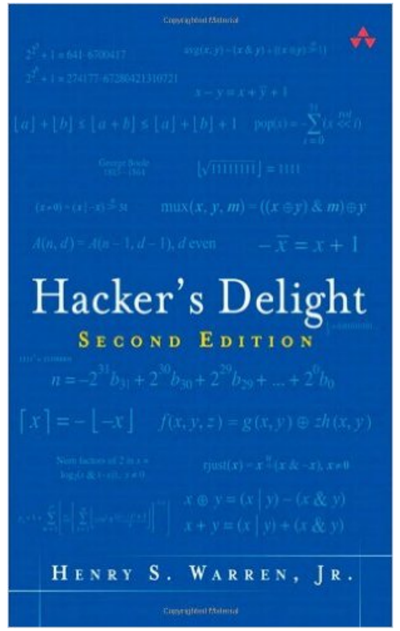
알고리즘 개요


Synthesis specification
Initial partial programs
Static Analysis expert
Abstract domains
Necessary Precondition Generator
Partial programs + Necessary preconditions
New partial programs
Composer
Components of size ≤ n
Bottom-Up Enumerator
Increase n
Failure: all partial programs tried
Complete Program
Success : I/O examples satisfied
Final program

예제 합성 문제

- 목표: 입력 비트벡터의 가장 오른쪽의 0 왼쪽에 있는 1들을 0으로 바꾸기
- 합성 대상 : f(x: BitVec) : BitVec.
- 생김새 조건:

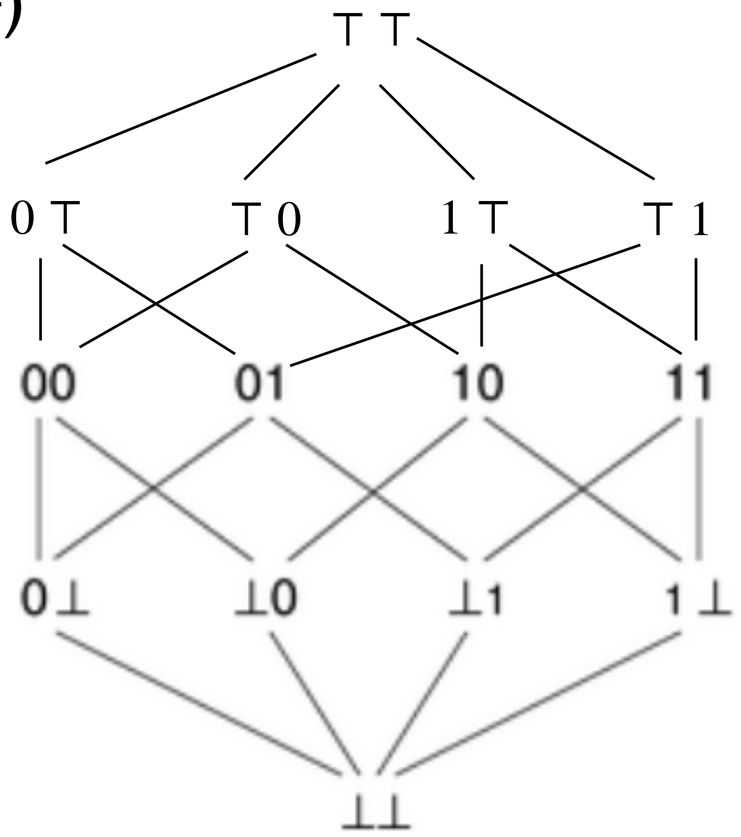
$$
\begin{array}{rcll}
S & \rightarrow & x \mid 0001_2 & \text{input bit-vector and bit-vector literals} \\
 & \mid & S \wedge S \mid S \vee S \mid S \oplus S & \text{bitwise logical binary operators} \\
 & \mid & S + S \mid S \times S \mid S/S \mid S >> S & \text{bitwise arithmetic binary operators} \\
 & & \ldots &
\end{array}
$$



- 제약 조건: $f(1011_2) = 0011_2$
- 해답: $f(x) = ((x + 0001_2) \oplus x) >> 0001_2$

정적 분석

- 실제 값 대신 "요약 값"으로 프로그램 실행을 모사(simulation)
  - 요약 : 빠짐없이 포섭 (예: 실제 값 : {0, 2, 6} → 요약 : 짝수)
- 비트필드 도메인
  - 각 비트를 $\{0,1, \bot, \top\}$ 중 하나로 표현
  - $\top$: 무슨 비트인지 모름, $\bot$: 존재하는 비트가 없음
- 예: $\top 01\top$은 집합 $\{0010_2, 0011_2, 1010_2, 1011_2\}$ 을 표현함
- 요약연산들은 #와 함께 표현할 예정
  - 예:, $1\top 10 \wedge^{\#} 00\top\top = 00\top 0$



미완성 프로그램 생성

- 하향식 탐색을 수행하여 미완성 프로그램들 집합 Q 생성
- 시작 비말단기호 S에서 시작하여 문법규칙을 반복적용하여 생성
- 이 예제에서는 아래 집합을 생성하였다고 가정:

$$Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$$

부품 생성

- 상향식 탐색을 수행하여 부품식 집합 C 생성
- 가장 작은 부품식들$(x, 0001_2)$부터 시작하여 이들을 조합하여 점차 더 큰 식들 생성
- 크기 n이하의 부품식들 생성 (맨 처음에는 $n = 1$)
- 이 때 입력에 대해서 같은 출력을 생성하는 부품식들 중 하나만 남기는 최적화 수행
  - 예: 0001_2 가 이미 부품식에 있는 상태에서 $x \wedge 0001_2$ 은 생성 안함
- 처음에는 $n = 1$ 이므로

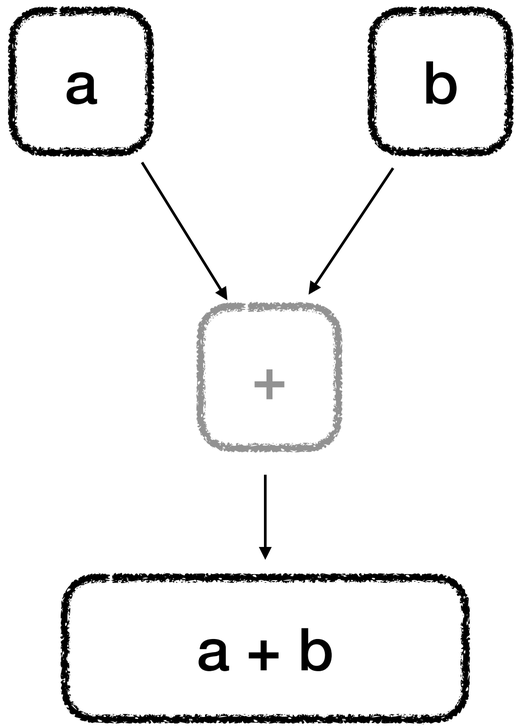
$$C = \{x, 0001_2\}$$

필요조건 분석

- 각 미완성 프로그램의 비말단기호마다 필요조건 유추
- 필요조건은 요약도메인의 한 원소로 표현됨
- 필요조건 추출을 위해 정방향, 역방향 분석 사용

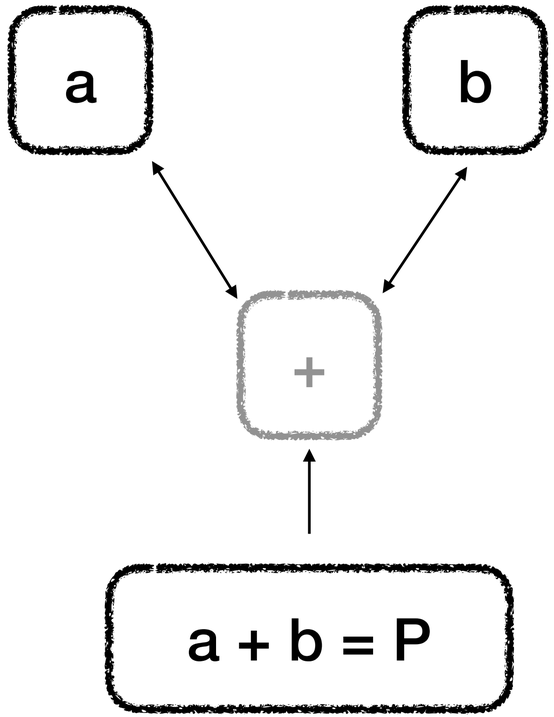
정방향(Forward) 및 역방향(Backward) 분석

정방향



a의 요약을 계산하고
b의 요약을 계산하고
그것으로 a+b의 요약을 계산

역방향



a+b가 만족해야할 조건 P가 주어졌을 때
a, b 각각의 요약이 만족해야할 조건을 계산
(a, b의 정방향 요약이 계산되어있으면 활용 가능)

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들:

코드 형태로:

```
o1 := ?
o2 := ?
o3 := o1 x o2
```

정방향 (forward) 비트필드 도메인 분석:

```
o1 := ?          TTTT
o2 := ?          TTTT
o3 := o1 x o2    TTTT
```

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들:

코드 형태로:

```
o1 := ?
o2 := ?
o3 := o1 x o2
```

정방향 (forward) 비트필드 도메인 분석:

```
o1 := ?          TTTT
o2 := ?          TTTT
o3 := o1 x o2    TTTT ⊓ 0011 = 0011
```

바람직한 출력과 교집합이 있으므로 싹수있음

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들:

코드 형태로:

```
o1 := ?
o2 := ?
o3 := o1 x o2
```

역방향 (backward) 비트필드 도메인 분석:

```
o1 := ?          TTTT
o2 := ?          TTTT
o3 := o1 x o2    TTTT ⊓ 0011 = 0011
```

바람직한 출력값 0011을 낼 수 있는 o1, o2 값 유추.
오버플로우 고려 시 임의의 값들이 될 수 있음.

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T\ T\ T\ T}, S_2 \mapsto \mathsf{T\ T\ T\ T}$

코드 형태로:

```
o1 := ?
o2 := ?
o3 := o1 x o2
```

역방향 (backward) 비트필드 도메인 분석:

```
o1 := ?          TTTT
o2 := ?          TTTT
o3 := o1 x o2    TTTT ⊓ 0011 = 0011
```

바람직한 출력값 0011을 낼 수 있는 o1, o2 값 유추.
오버플로우 고려 시 임의의 값들이 될 수 있음.

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T\ T\ T\ T}\ , S_2 \mapsto \mathsf{T\ T\ T\ T}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 ⊕ o2
o4 := o3 >> 0001₂
```

정방향 (forward) 비트필드 도메인 분석:

```
o1 := ?                TTTT
o2 := x                1011
o3 := o1 ⊕ o2          TTTT
o4 := o3 >> 0001₂      TTTT
```

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T\ T\ T\ T}, S_2 \mapsto \mathsf{T\ T\ T\ T}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 ⊕ o2
o4 := o3 >> 0001₂
```

정방향 (forward) 비트필드 도메인 분석:

```
o1 := ?                TTTT
o2 := x                1011
o3 := o1 ⊕ o2          TTTT
o4 := o3 >> 0001₂      TTTT ⊓ 0011 = 0011
```

바람직한 출력과 교집합이 있으므로 싹수있음

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) >> 0001_2, (S_4/x) >> 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T\ T\ T\ T}, S_2 \mapsto \mathsf{T\ T\ T\ T}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 ⊕ o2
o4 := o3 >> 0001₂
```

역방향 (backward) 비트필드 도메인 분석:

```
o1 := ?          TTTT
o2 := x          1011
o3 := o1 ⊕ o2    011T
o4 := o3 >> 0001₂  0011
```

011T >># 0001 = 0011

필요조건 분석

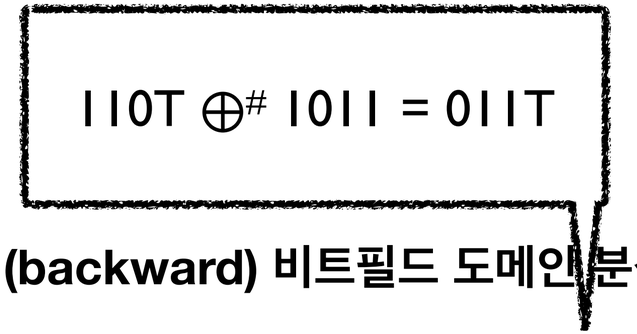
- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{TTTT}, S_2 \mapsto \mathsf{TTTT}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 ⊕ o2
o4 := o3 >> 0001₂
```

역방향 (backward) 비트필드 도메인 분석:

```
o1 := ?          110T
o2 := x          1011
o3 := o1 ⊕ o2    011T
o4 := o3 >> 0001₂ 0011
```



필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{TTTT}, S_2 \mapsto \mathsf{TTTT}, S_3 \mapsto 110\,\mathsf{T}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 ⊕ o2
o4 := o3 >> 0001₂
```

역방향 (backward) 비트필드 도메인 분석:

o1 := ?	110T
o2 := x	1011
o3 := o1 ⊕ o2	011T
o4 := o3 >> 0001₂	0011

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_2 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_3 \mapsto 110\ \mathsf{T}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 / o2
o4 := o3 >> 0001₂
```

정방향 (forward) 비트필드 도메인 분석:

```
o1 := ?                TTTT
o2 := x                1011
o3 := o1 / o2          TTTT
o4 := o3 >> 0001₂      TTTT ⊓ 0011 = 0011
```

바람직한 출력과 교집합이 있으므로 싹수있음

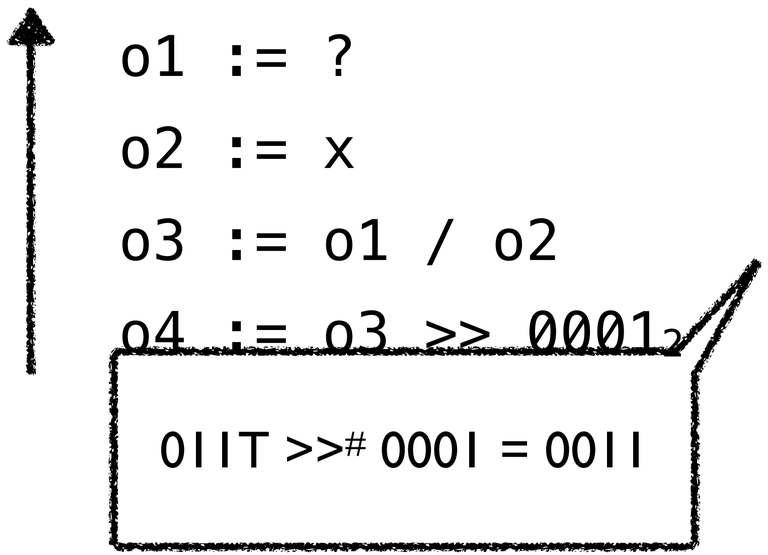
필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T\ T\ T\ T}, S_2 \mapsto \mathsf{T\ T\ T\ T}, S_3 \mapsto 110\ \mathsf{T}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 / o2
o4 := o3 >> 0001₂
```

역방향 (backward) 비트필드 도메인 분석:



필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{TTTT}, S_2 \mapsto \mathsf{TTTT}, S_3 \mapsto 110\,\mathsf{T}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 / o2
o4 := o3 >> 0001₂
```

역방향 (backward) 비트필드 도메인 분석:

```
o1 := ?                TTTT
o2 := x                1011 ⊓ 00TT = ⊥011
o3 := o1 / o2          011T
o4 := o3 >> 0001₂      TTTT ⊓ 0011 = 0011
```

o1 / o2 가 011T이 되어야함. o1은 최대 1111_2 (즉 15)를 값으로 가질 수 있음
011T는 $\{0110_2, 0111_2\}$ (즉 6과 7)을 의미. o2의 가능 값은 0001_2과 0010_2 (즉 00TT)

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{TTTT}, S_2 \mapsto \mathsf{TTTT}, S_3 \mapsto 110\,\mathsf{T}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 / o2
o4 := o3 >> 0001₂
```

역방향 (backward) 비트필드 도메인 분석:

```
o1 := ?                TTTT
o2 := x                1011 ⊓ 00TT = ⊥011
o3 := o1 / o2          011T
o4 := o3 >> 0001₂      TTTT ⊓ 0011 = 0011
```

그런데 o2의 요약값은 이미 1011인 상태. 00TT이 될 수 없음.

그러므로 이 미완성 프로그램은 싹수없음!

필요조건 분석

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2, (S_4 / x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_2 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_3 \mapsto 110\ \mathsf{T}$

코드 형태로:

```
o1 := ?
o2 := x
o3 := o1 / o2
o4 := o3 >> 0001₂
```

역방향 (backward) 비트필드 도메인 분석:

`o1 := ?`	`TTTT`
`o2 := x`	`1011 ⊓ 00TT = ⊥011`
`o3 := o1 / o2`	`011T`
`o4 := o3 >> 0001₂`	`TTTT ⊓ 0011 = 0011`

부품 조합

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T\ T\ T\ T}, S_2 \mapsto \mathsf{T\ T\ T\ T}, S_3 \mapsto 110\ \mathsf{T}$

$S_1 \times S_2$

출력값의 요약이 TTTT인 부품을 넣어봄

부품 조합

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_2 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_3 \mapsto 110\ \mathsf{T}$

$$x \times S_2$$

부품 조합

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_2 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_3 \mapsto 110\ \mathsf{T}$

$$x \times S_2$$

S2의 필요조건 유추. S2의 (모르겠는) 출력값을 y라고 할때 다음 식을 만족하는 y를 구하면 됨

$$1011_2 \times y = 0011_2 \mod 16$$

확장된 유클리드 알고리즘을 사용하면 $y = 1001_2$ 임을 알 수 있음

부품 조합

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T\ T\ T\ T}, S_2 \mapsto \mathsf{T\ T\ T\ T}, S_3 \mapsto 110\ \mathsf{T}$

~~$x \times S_2$~~

출력값의 요약이 1001인 부품을 찾아봄. 없음.

부품 조합

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T\ T\ T\ T}, S_2 \mapsto \mathsf{T\ T\ T\ T}, S_3 \mapsto 110\ \mathsf{T}$

$$(S_3 \oplus x) \gg 0001_2$$

출력값의 요약이 110T인 부품을 찾아봄. 없음.

부품 조합

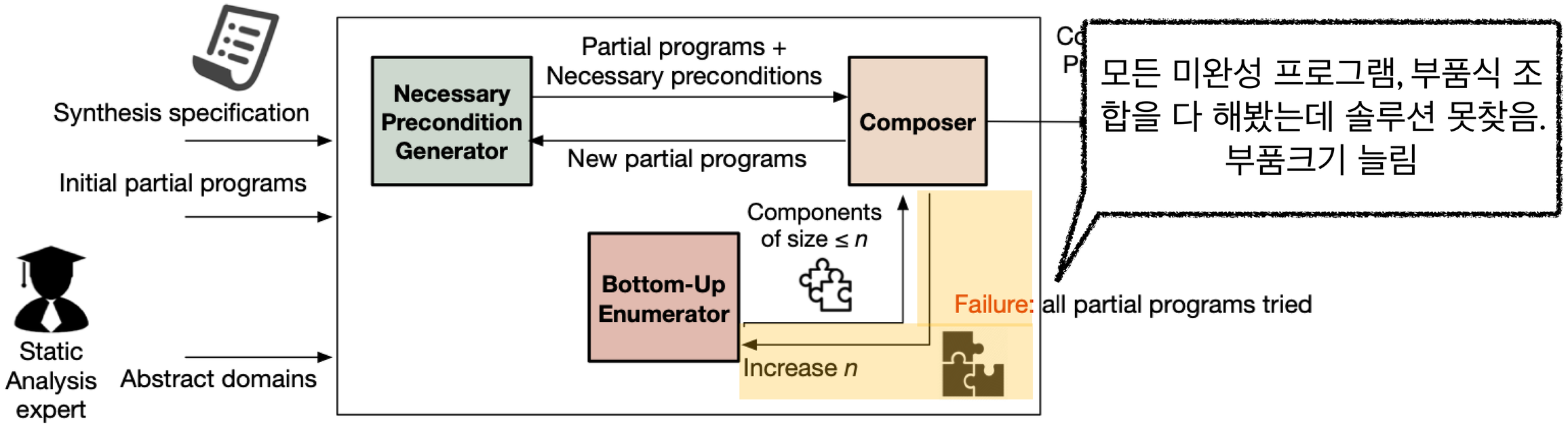
- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T\ T\ T\ T}, S_2 \mapsto \mathsf{T\ T\ T\ T}, S_3 \mapsto 110\ \mathsf{T}$

~~$(S_3 \oplus x) \gg 0001_2$~~

출력값의 요약이 110T인 부품을 찾아봄. 없음.

부품 조합

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2\}$
- 필요조건들: $S_1 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_2 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_3 \mapsto 110\ \mathsf{T}$



부품 늘리기

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2, x + 0001_2, \cdots\}$
- 필요조건들: $S_1 \mapsto \top\ \top\ \top\ \top, S_2 \mapsto \top\ \top\ \top\ \top, S_3 \mapsto 110\ \top$

크기 3 이하의 부품들 생성

부품 조합

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2, x + 0001_2, \cdots\}$
- 필요조건들: $S_1 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_2 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_3 \mapsto 110\ \mathsf{T}$

$$(S_3 \oplus x) \gg 0001_2$$

출력값의 요약이 110T인 부품을 찾아봄.
있음! $x + 0001_2$

부품 조합

- 제약조건: $f(1011_2) = 0011_2$
- 미완성 프로그램: $Q = \{(S_1 \times S_2, (S_3 \oplus x) \gg 0001_2\}$
- 부품식 집합: $C = \{x, 0001_2, x + 0001_2, \cdots\}$
- 필요조건들: $S_1 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_2 \mapsto \mathsf{T}\ \mathsf{T}\ \mathsf{T}\ \mathsf{T}, S_3 \mapsto 110\ \mathsf{T}$

$$((x + 0001_2) \oplus x) \gg 0001_2$$

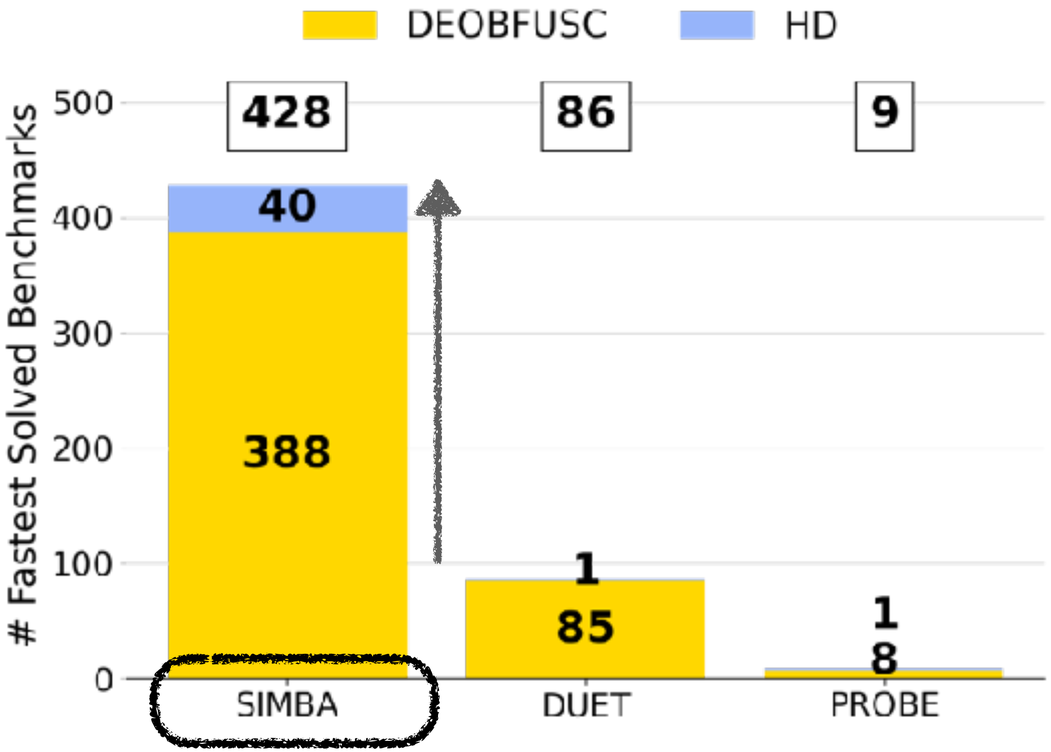
해당 부품을 껴 넣어봄 ⇨ 솔루션

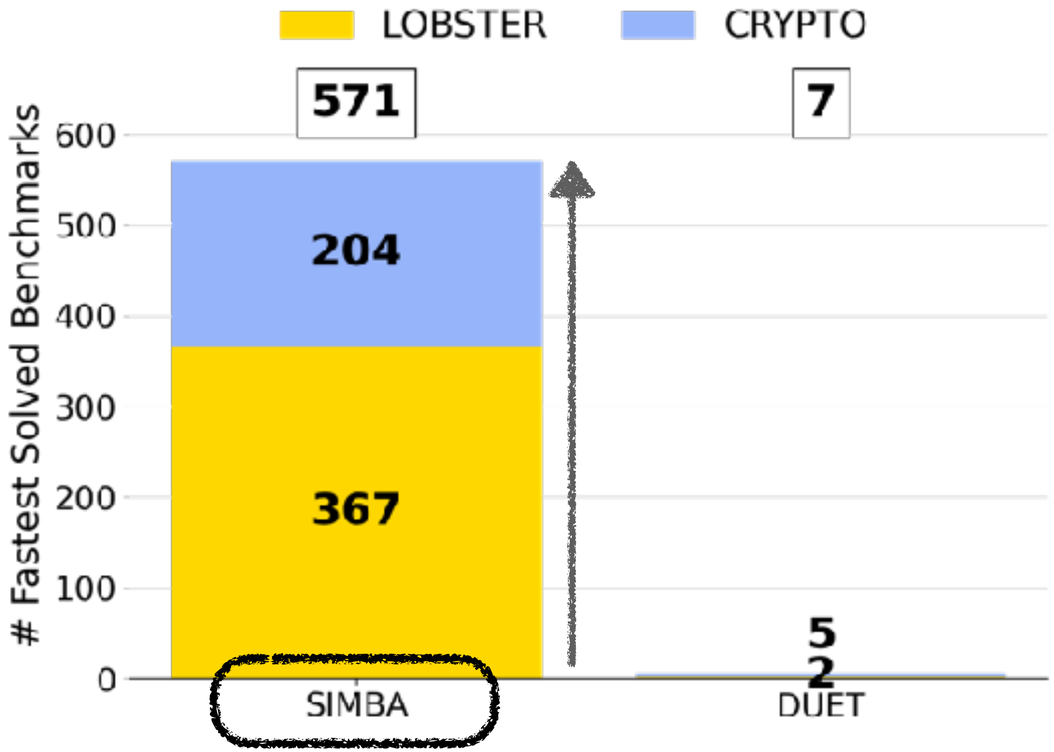
실험

- 구현체: **Simba**
- 벤치마크: 총 1,125개 프로그램 합성 경진대회에 사용된 문제들
  - HD: hacker's delight 문제 44개
  - Deobfsc: QSynth VR-EA 데이터셋 500개
  - Lobster: 369 회로 최적화 문제 (동형암호 가속화)
  - Crypto: 212 회로 최적화 문제 (부채널 공격 회피)
- 비교대상 합성기들
  - **duet**: Woosuk Lee, "Combining the Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis", POPL'21
  - **probe**: Barke et al., Just-in-Time Learning for Bottom-Up Enumerative Synthesis, OOPSLA'20

실험결과

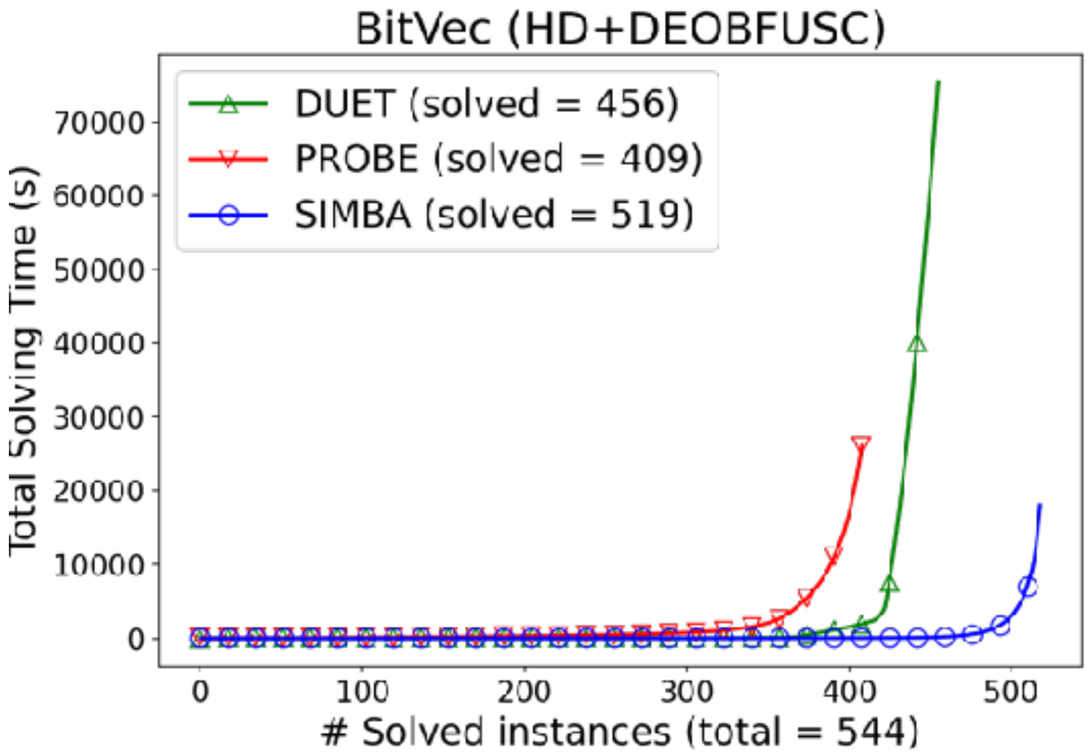
- 타 도구 대비 압도적인 성능

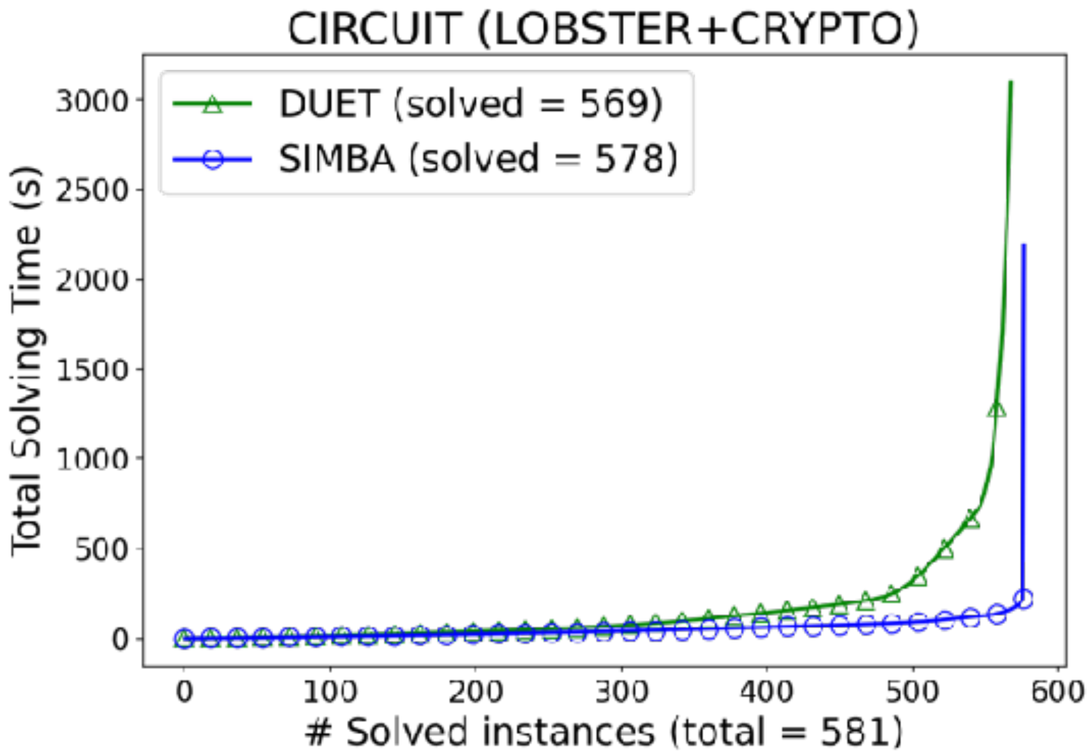




실험결과

- 타 도구 대비 압도적인 성능





결론

- 자동탐색 시스템을 사용하면
  - 현대 컴퓨터의 높은 성능을 활용
  - 프로그램 합성 (Program Synthesis), 동일식 모으기 (Equality Saturation) 등
- 특정한 경우에는
  - 사람이 다루기 힘든 영역에 대해
  - 논리회로, 비트정수 등 저수준 언어
- 전문가의 수동 알고리즘보다 나은 결과를 낼 수 있다.
  - 새로운 최적화 규칙 발견
  - 최적의 결과를 내는 기기묘묘한 규칙 적용순서 발견